THE UNIVERSITY
of EDINBURGH

Multi-Tasking Scheduling for Heterogeneous Systems

Yuan Wen

Doctor of Philosophy

Institute of Computing Systems Architecture

School of Informatics

University of Edinburgh

2017

Abstract

Heterogeneous platforms play an increasingly important role in modern computer systems. They combine high performance with low power consumption. From mobiles to supercomputers, we see an increasing number of computer systems that are heterogeneous.

The most well-known heterogeneous system, CPU+GPU platforms have been widely used in recent years. As they become more mainstream, serving multiple tasks from multiple users is an emerging challenge. A good scheduler can greatly improve performance. However, indiscriminately allocating tasks based on availability leads to poor performance. As modern GPUs have a large number of hardware resources, most tasks cannot efficiently utilize all of them. Concurrent task execution on GPU is a promising solution, however, indiscriminately running tasks in parallel causes a slowdown.

This thesis focuses on scheduling OpenCL kernels. A runtime framework is developed to determine where to schedule OpenCL kernels. It predicts the best-fit device by using a machine learning-based classifier, then schedules the kernels accordingly to either CPU or GPU. To improve GPU utilization, a kernel merging approach is proposed. Kernels are merged if their predicted co-execution can provide better performance than sequential execution. A machine learning based classifier is developed to find the best kernel pairs for co-execution on GPU. Finally, a runtime framework is developed to schedule kernels separately on either CPU or GPU, and run kernels in pairs if their co-execution can improve performance. The approaches developed in this thesis significantly improve system performance and outperform all existing techniques.

Lay Summary

In the past decade, processors originally designed for graphic processing have also been widely used to solve general purpose computational problems. Architectural differences between the Graphic Processing Unit (GPU) and the Central Processing Unit (CPU) determine that, in most cases, either of them works better for a particular application than the other. As an open standard, OpenCL provides programmers with a method to program an application once but run it on various processors.

As it is widely known, OpenCL applications are functionally portable; however, they are not performance portable across different processors. Therefore, finding out the most appropriate device for a given program is significant, especially in a multitasking environment. Also, since GPU vendors increase hardware resources in their products of every generation, GPU utilisation performs a critical role in system throughput. Effective sharing a GPU by multiple OpenCL kernels enhances overall performance by running these kernels concurrently without any significant interference from each other while ineffective sharing behaves oppositely.

In this thesis, we focus on a smart scheduling method for OpenCL kernels in a multitasking environment to improve the system performance. In our targeting system, both multi-core CPU and discrete GPU are candidate scheduling devices. The goal is to find out the most appropriate target device for every single kernel and all the kernel pairs if they can benefit from their co-execution on the same GPU. We integrate machine learning and just-in-time compilation technique to our runtime framework to carry out this smart scheduling.

Acknowledgements

I would like to thank my academic advisor, Professor Michael F.P. O'Boyle, for his constant support. He guided me how to do the research and make things done efficiently. I feel very lucky to be his PhD student. I have learned a lot by working with him over the past years. His invaluable advice would keep me going for further success in my future career.

I would like to thank my friend and colleagues in the CArD research group. In particular, I would like to thank Alberto Magni, Thibaut Lutz, Kiran Chandramohan, Cheng-Chieh Huang, Karthik Bharghava Rajaram and Zheng Wang who helped me by sharing their knowledge in academic research and experience in engineering. I would like to thank my officemate Alex Collins, Dominik Grewe, Siddharth Mohanty, Juan Jose Fumero, Arpit Joshi, and Paschalis Mpeis. I am also grateful to Kuba Kaszyk who shared his desk with me when I was doing my thesis writing. I would also like to thank Chris Fensch, Erik Tomusk, Harry Wagstaff, Priyank Faldu, Marco Elver, Volker Seeker, Tom Spink, George Stefanakis, Murali Krishna Emani, Tobias Edler von Koch, Stephen Kyle, Andrew J. McPherson, Konstantina Mitropoulou, Vasileios Porpodas, Karthik T. Sundararajan, and all good friends and colleagues in CArD group.

I would like to thank Hedley Francis, Anton Lokhmotov and ARM for the support in my PhD research.

I would like to thanks my old friends who we keep in touch all the past years and share scientific ideas. In particular, I would like to thank Like Yan, Bin Xie, and Huanjie Xu.

Finally, and most importantly, I am very grateful to my parents for their constant and unconditional support.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Some of the material used in this thesis has been published in the following papers:

- Yuan Wen, Zheng Wang, Michael O'Boyle, *Smart Multi-Task Scheduling for OpenCL Programs on CPU/GPU Heterogeneous Platforms*, In the 21st annual IEEE International Conference on High Performance Computing (HiPC 2014)

- Yuan Wen,Michael O'Boyle, *Merge or separate? Multi-job scheduling for OpenCL Kernels on CPU/GPU Platforms*, In Proceedings of the Workshop on General Purpose Processing Using GPUs (GPGPU 2017)

(*Yuan Wen*)

Table of Contents

Chapter 1

Introduction

Modern computer systems are becoming increasingly heterogeneous as they provide high performance with low energy consumption. Such systems consist of more than one type of processor, e.g. central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs) and other types of application-specific integrated circuits (ASICs). At least one processor is a CPU which acts as the host device to run the operating system (OS), while the others typically work as accelerators to speed up a particular part of the workload. By allocating programs to separate processors, which have different architectures, heterogeneous systems can provide flexible choices to meet different performance and energy budgets. CPU and accelerators can be placed into the same integrated circuit (IC) or, more commonly, into distinct ICs which are then interconnected via buses, e.g. PCI Express.

The most popular heterogeneous system, CPUs+GPUs hybrid system has been widely used in mobile platforms, desktop environments, supercomputers, and data center. Typically it consists of a multi-core CPU and a discrete graphics device that has a GPU processor and private memory. The multi-core CPU can work both as host and an accelerator. As the host processor it supports the OS, and controls the parallel workload distribution and memory copying. It acts as an accelerator when carrying out parallel workload computation. Unlike CPUs, GPUs always perform as accelerators. When both CPU and GPU work as accelerators, the difference in their architectures affects the performance of workloads.

The architecture of a multi-core CPU is designed to provide high performance for workloads that have strong data locality and complex control flows. GPUs consist of many simpler processing units and require a new programming language to exploit their parallel structure. CUDA [NVIDIA (2016)] and OpenCL [Khronos (2016)]

are the most popular frameworks, which are developed and supported by Nvidia and Khronos Group separately. Both of them ship with C-like programming languages (and corresponding runtime compilers) and a set of application programming interfaces (APIs) which allow workload and memory management. An application, under both frameworks, is divided into two parts: the host code and the device code. The device code, also called the kernel, is the computation workload that is executed by the accelerator, which is also known as the compute-device, or device. The host code executes on the CPU processor (or host-device) and issues kernels to the accelerator and manages memory for both host- and compute- devices. The main difference between CUDA and OpenCL is that CUDA only supports CUDA-enabled GPUs from Nvidia, but OpenCL, as an open standard, embraces a wide range of processors, such as CPU, GPU, DSP, and field-programmable gate arrays (FPGAs). Therefore, in this thesis, we focus on OpenCL as it is portable across different processors.

The portability of OpenCL kernels provides an opportunity for workload distribution on CPU-GPU platforms. A kernel can be either allocated to the CPU or GPU. The performance of a kernel varies on different processors; some kernels have reduced execution time on a GPU relative to the CPU, but some kernels experience the opposite. Some examples of this are shown in Figure 4.2 a and 6.1 a. Conventionally, the GPU is designed to serve a single kernel at a time. However, as GPU becomes mainstream, multiple workloads from multi-users are increasingly common in this type of system, but there is no well-designed support for multi-task management. The default method simply maps a task whenever an idle device is found. This method does not consider the differences between kernel on different devices, and therefore, a First Come First Served (FCFS) method can cause sub-optimal throughput of the overall system. Also, serving one kernel at a time usually leads to GPU underutilization, especially for GPUs that have a large number of hardware resources to support high parallelism computing. To improve the performance of CPU-GPU heterogeneous platforms, we have to overcome the two above challenges in a multi-task environment.

1.1 Multi-tasking Challenges

When scheduling multiple tasks on CPU-GPU platform, two main challenges have to be addressed: device affinity and GPU utilization. Device affinity describes the performance variation of single kernels on different devices. Utilization identifies to what extent various types of hardware resources have been efficiently used. Both device

affinity and utilization are critical to performance as the GPU is designed as an accelerator for a single application and has little support for discriminating and sharing amongst workloads.

Device Affinity

OpenCL kernels can be executed on either multi-core CPU or GPU, and therefore, at runtime, the scheduler has to decide which kernel is allocated to which device. Architectural differences between CPU and GPU determines kernel execution time on them. Some kernels run faster on the GPU; while others perform better on the CPU, especially when taking into consideration of the data movement between CPU main memory and GPU memory via PCI Express.

Because some kernels perform well on CPU, and some on GPU, it is vital for the system performance to allocate kernels to their best-fit device. Also, due to each kernel's different device affinity, multi-core CPU and GPU can serve separate kernels in parallel, and hence, improve the system throughput further.

GPU Utilization

Because of the GPU's massive parallel architecture, it provides the potential for excellent performance as long as there is no complex control flow within the workload. Therefore, kernels are often designed with few branches to satisfy this requirement. With hardware technology improvement, an increasing number of various hardware resources, such as computing unit, register, cache, and special units are integrated into the GPU processor. A consequence of this trend is that a simple kernel is unlikely to use all types of resources and can not share the extra resources with others since most of the GPU processors work as a designated device that serves only one kernel at a time.

1.2 Contribution

This thesis presents solutions for the challenges described above. A runtime framework is developed to detect OpenCL kernels device affinities. It predicts the best-fit device by using a machine learning based classifier, then inserts the kernels, according to their estimated best device, into a task queue from which they can be issued to either CPU or GPU from the corresponding end of the queue. To improve GPU utilization,

a kernel merging approach is proposed. Here kernels are merged if their predicted co-running can provide better performance than their sequential execution. Since random co-running kernels may improve utilization but not necessarily improve performance, a machine learning based classifier is developed to find out the best kernel pairs for co-execution on GPU according to their code features and runtime parameters. Finally, a runtime framework is developed to schedule kernels separately on either CPU or GPU, and run kernels in pairs if co-running can improve the performance.

The following list separately summarizes the contributions of this thesis:

Chapter 4: The first contribution of this thesis is the development of a machine learning based individual kernel scheduler. By learning from the code features and runtime parameters, an off-line model is trained which classifies each new arriving kernel as a CPU or GPU friendly task. The scheduler inserts the kernels according to their device affinities into the task queue. GPU friendly tasks are queued to one end of the queue, while CPU friendly kernels are queued towards the other end of the queue. The scheduler dynamically issues a kernel from either end of the task queue when the appropriate device is idle. The performance is improved by running a kernel on the most appropriate device as well as by concurrent kernel execution on CPU and GPU. Comparing to a random scheduling method our approach improves the throughput by 20%+ and optimizes the average turnaround time by 55%+.

Chapter 5: The second contribution of this thesis is a machine learning based concurrent kernel pairs selection strategy. Co-running kernels can improve GPU hardware resource utilization; but do not necessarily mean improved performance. We developed a machine learning based approach to dynamically select kernel pairs that can profitably be co-executed. A machine learning model is trained off-line but used online to pair up newly arriving kernels according to their features. A graph based scheduling approach is designed to maximize the number of kernel pairs, so as to increase the overall throughput. The experiment results show that our method improves the performance by 10%+ and 20%+ over two state-of-the-art approaches.

Chapter 6: The third contribution of this thesis is the design of a runtime framework that co-schedules both separate and combined kernels. The runtime framework integrates both separate and concurrent kernel classifiers. By setting up a global context on all OpenCL supported devices, the framework manages the hardware resources and

kernel allocations. All OpenCL applications are registered with the framework, and therefore, the framework manages data and issues the kernel on behalf of each application. This runtime layer can schedule kernels separately to CPU and GPU, or in pairs to GPU, with the help of machine learning based classifiers. As the framework works globally to serve all applications, there is no need for each program to initialize the environment, and therefore, it reduces the execution time further. The experiment shows that co-scheduling method improves the performance by 50%+ comparing with state-of-the-art concurrent kernel executions and 20%+ better than the scheduling approach proposed in chapter 4.

1.3 Thesis Outline

The remainder of this thesis is organised as follows:

Chapter 2: This chapter presents the technical background used in the remainder of the thesis. It first introduces heterogeneous systems and two typical processors that are widely used in heterogeneous platforms: CPU and GPU. Secondly, it introduces the OpenCL programming model and memory hierarchy. Next, this chapter gives a brief summary of machine learning techniques. A machine learning method has been used to classify the candidate kernels based on their features. Finally, benchmarks and the evaluation method used in this thesis are described.

Chapter 3: This chapter discusses related prior work. It first discusses the scheduling methods on the heterogeneous system, which include allocating a single task to multiple devices, multiple tasks to the single device, and multiple tasks to multiple devices. It then introduces optimization on communication between host device and accelerators. The method for analyzing software performance on the heterogeneous platform is then introduced. Finally, this chapter discusses machine learning based approaches for heterogeneous systems.

Chpater 4: This chapter introduces a machine learning based runtime scheduler which schedules individual OpenCL kernels to their most appropriate devices. The model is trained off-line on code features and runtime parameters of training kernel samples. The trained model is a classifier that labels the new arriving kernels as CPU or GPU friendly tasks. The scheduler inserts the GPU preferable tasks to one end of the queue

while inserting the CPU preferable tasks towards the other end. At runtime, tasks are dequeued from either end of the queue as long as the connected device is idle. The scheduler can issue tasks to their best-fit device dynamically.

Chapter 5: This chapter introduces the construction of a pair of concurrent kernels by merging their source code. Compared to issuing via separate command queues, this merged kernel is not affected by the runtime environment or GPU hardware scheduler. A machine learning based classifier is presented to detect whether or not two kernels co-running have a better performance than running them sequentially. Finally, a graph based scheduling method is proposed to maximize the number of kernel pairs.

Chapter 6: This chapter introduces the design of our runtime framework and a scheduling method that combines the techniques developed in Chapter 4 and Chapter 5. Using this runtime layer, OpenCL programs register with the framework, which launches the kernels separately or pairwise with the support of machine learning based classifiers. All tasks share a common context and hence reduce runtime overhead.

Chapter 7: This chapter concludes this thesis. It first summarizes the contributions of the work in each technical chapter. Next, a critical analysis is presented discussing the limitation of this work. Finally, it describes some areas of future work.

Chapter 2

Background

This chapter provides the necessary background material for the remainder of this thesis. The key topics covered in this chapter are heterogeneous systems and machine learning.

2.1 Heterogeneous System

Heterogeneous systems refer to platforms that comprise more than one type of processor. These systems have the potential to improve the performance and energy efficiency by having different processors. There are two types of heterogeneous systems depending on the instruction set architecture (ISA): single-ISA and heterogeneous-ISA.

Single-ISA systems consist of multiple CPU cores. The cores share the same instruction set but have different hardware micro-architecture configurations. A typical example of a single-ISA system is ARM's big.LITTLE. As shown in figure 2.1, it consists of compute-efficient Cortex-A15 CPUs and a power-efficient Cortex-A7 CPUs. Executing task can be swapped between these cores, on the fly, to improve performance or energy consumption.

Heterogeneous-ISA systems consist of processors with different architectures. Typically, these systems consist of a central processing unit (CPU) and one or more specialized processors. The specialized processors are accelerators that are designed for a particular kind of computation. The CPU is used to execute the operating system and deploy workloads to the appropriate accelerators. Heterogeneous-ISA systems normally significantly outperform the best single-ISA model [Venkat and Tullsen (2014)], which make it more attractive in heterogeneous computing.

The most widely known heterogeneous-ISA model is the CPU-GPU architecture.

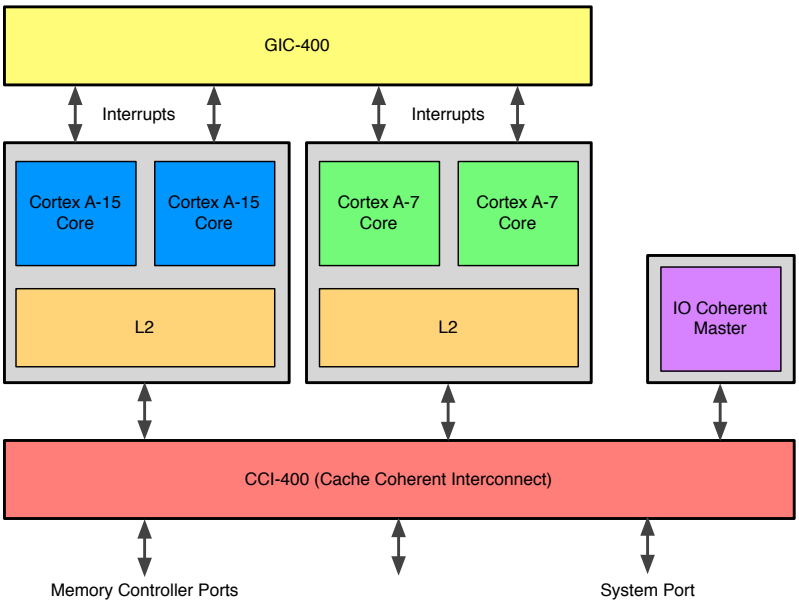


Figure 2.1: big.LITTLE system. The big core, Cortex-A15, is designed to provide high performance because it has a complex pipeline and more instruction parallelism. The little core, Cortex-7, is power efficient, as its pipeline is relatively simple. Both big and little cores share the same ISA.[ARM (2016)]

It has been widely used in mobile, desktops, and data centre, because of its high computing performance delivered per watt. There are two flavours of CPU-GPU system: integrated and discreted. An integrated GPU is placed in the same circuit package with the CPU. It does not have independent RAM; instead, it shares the CPU's main memory. Discrete GPU works as a stand-alone accelerator which accesses data from its designated RAM. Though integrated GPU consumes less power, this thesis focuses on systems that consist of discrete GPUs, as they provide higher performance than their integrated counterpart and have been more widely used in desktops and supercomputers.

2.1.1 Central Processing Units

The central processing unit (CPU) executes computer programs by performing arithmetic, control and input/output (I/O) operations. In each generation of CPU, the primary goal of its designers has been to make it process instructions faster. Much effort has been put in improving instruction-level parallelisms (ILP) by using out-of-order execution, superscalar execution, branch prediction, and instruction pipelining. Considerable effort has also been given to increasing the CPU clock frequency, as a higher

clock frequency increases performance. However, this trend has stopped due to the power wall [Venu (2011), Chung et al. (2010), Brodtkorb et al. (2010), Meenderinck and Juurlink (2008)].

As single processors have now reached peak clock frequency, the major CPU manufacturers, such as Intel and AMD, have switched their strategy to multi-cores. By integrating more cores onto a single integrated circuit die, they provide enhanced performance by increasing the throughput of the system. Typical multi-core processors that have been widely used include, e.g. AMD Phenom II X2 and Intel Core Due, AMD Phenom II X4 and Intel Core i5 and Core i7. Cores in these systems normally have the same architecture as their single-processor counterparts.

Though modern CPUs support SIMD (Single Instruction, Multiple Data) processing, they are mainly designed to provide high performance for the workload that has strong data locality and complex control flows. For workloads that have large parallelism and simple control flow, the GPU is a good alternative device, compared to its CPU counterpart it has a strengthened capability for SIMD processing.

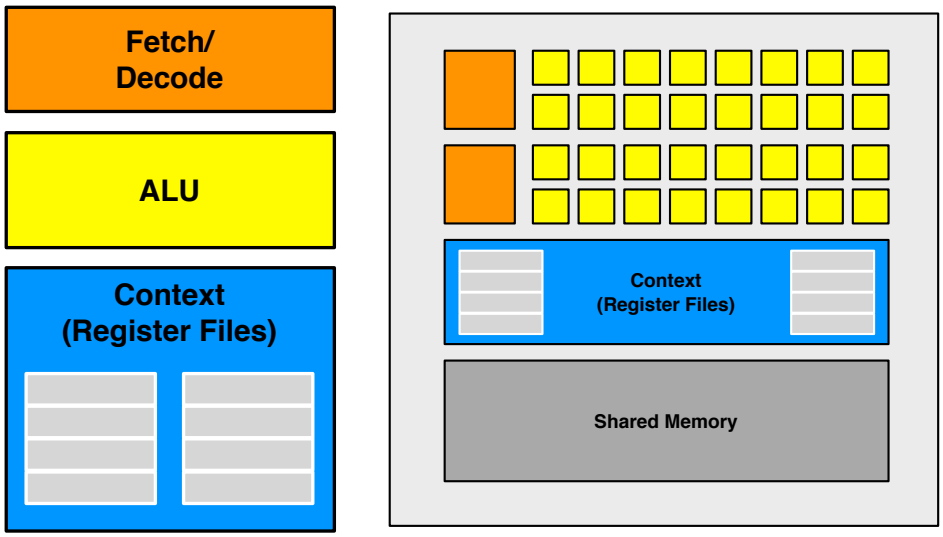
2.1.2 Graphic Processing Units

Graphics processing units (GPU) are specialized processors that originally targeted accelerated graphics manipulation and display. They are an extensively use in personal computers, workstations, and game consoles. They are designed to have a highly parallel structure, which makes them efficient at processing videos and images. This feature is also attractive to a large number of more general-purpose applications, particularly those scientific programs, which have high parallelism.

General Architecture

The core architecture difference between CPU and GPU is that the GPU has large numbers of simple cores to exploit massive parallelism via SIMD processing while the CPU has sophisticated circuit logic to tackle complex control flows. To highlight the differences between CPUs and GPUs architecture simplified diagrams are shown in figure 2.2.

As the basic unit, the arithmetic logic unit (ALU) performs the arithmetic and logic operations on both CPU and GPU. The ALU acquires its input from registers (or main memory) and writes the results back to registers (or main memory) when executing the instruction which is fetched and decoded by the hardware logic. For a



(a) Simplified CPU Model.

(b) Simplified GPU Model.

Figure 2.2: Simplified models for CPU and GPU. To highlight the architectural difference, the sophisticated control and connection logics are removed from the model. Both of the models are simple cores. In practice, multiple cores are placed in the real circuit package, particular for GPU processors. [Fatahalian (2008)]

simplified CPU model, every instruction processes one data at a time and this shown in figure 2.2a by connecting single ALU to a Fetch/Decode module. For the GPU model, each instruction is decoded and performed on a number of different data. In figure 2.2b, this mechanism is shown by connecting multiple ALUs to a single Fetch/Decode unit.

This SIMD architecture ensures GPU processors are efficient for the applications that have high parallelism. However, it also enforces some constraints on the applications to perform efficiently.

Divergence Sensitiveness

SIMD works well when executing linear instruction sequences, but experiences poor performance when encountering divergence control flow. When ALUs reach a branch instruction, they compare control flow results with the condition. For those that meet the condition, they perform the same statements and leave the others to execute a different group of instructions. As a result, when a branch occurs, only a subsection of ALUs are active at any given time, making the overall throughput suboptimal. Figure 2.3 shows an example of this problem.

In the example in figure 2.3, we assume 8 ALUs execute the same instruction

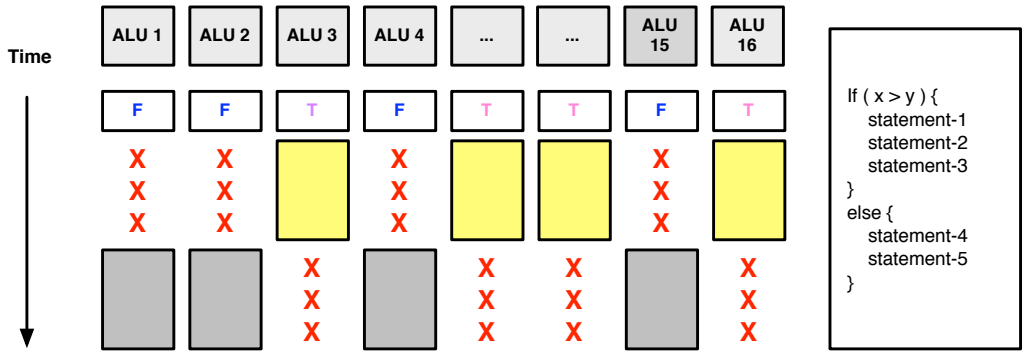


Figure 2.3: Divergence impacts on SIMD processor. In the worst case, 15 ALUs become idle because of the branch. On average, code divergence would hurt SIMD performance by 50%

synchronously at any given time. When the program reaches a branch instruction, e.g. if-statement in this example, the ALUs compare the condition of the branch. In this example, all ALUs detect whether or not their local value of x is greater than y. For the ALUs that have a result of x greater than y, they carry on with the statements 1 to 3 in parallel, while the others have to wait until those subsets of ALUs finishing their computing and then carry on executing the statements of 4 and 5 synchronously. Therefore, the branch instruction causes ALUs to diverge, only a subset of them that meet the same condition can work in parallel. In this example, due to the branch, the throughput of the GPU has dropped by 50%.

Divergence sensitivity means simple control flow is preferable. However, the CPU counterpart is not constrained by this, as it does not have a massive SIMD working model.

Memory Accessing Latency Sensitivity

Memory access also has a significant impact. When the data is not available in the register or local cache, the processor has to issue a memory access to fetch the data. Loading data from main memory can cost hundreds of ALU cycles. As all ALUs are working synchronized, they all become idle until data arrives in the corresponding registers. To bypass the stall caused by memory access, the SIMD processors are equipped with large register files so as to allow multiple sets of threads.

As each ALU under the same Fetch/Decode module processes the same instruction on different data independently, the linear instruction sequence on each ALU can be viewed as a thread. In figure 2.4, 16 ALUs share a Fetch/Decode module, hence in

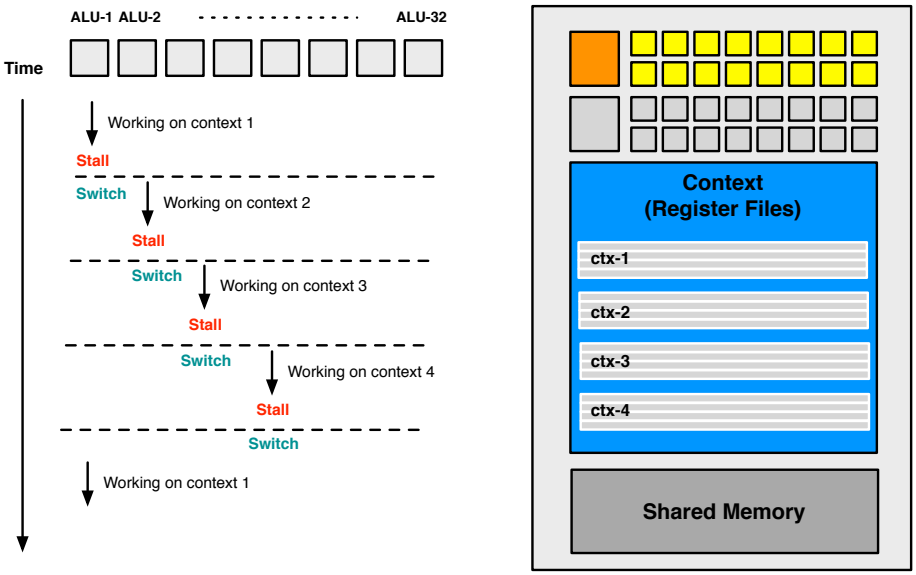


Figure 2.4: Context switch. Long latency memory access may cause threads stall. Hardware scheduler can switch to another set of threads while the current one is waiting for the memory operation. Switching between different sets of thread hides the latency of memory accessing. [Fatahalian (2008)]

this example, 16 threads can run synchronously at the same time and threads set size is 16. Therefore, if there were multiple thread sets, once the memory stall happens in one set the ALUs can perform another set of threads while they wait for the data accessing from the memory. Each set of threads has a context which consists of a group of registers that contains the status and thread data. Modern GPU processors equipped with large register files maintain multiple contexts so as to hide the memory latency by switching between thread sets.

Figure 2.4 shows how thread sets switching hides memory access overhead. Once a thread within the set can not access its data via the cache, it issues a long latency instruction to access the main memory. Other threads within the same set also have to stall. At this time, the hardware scheduler checks whether or not there exists another set of threads that can be switched in and carry on their execution. As long as such a candidate thread set is found, it will be switched in and executed until one of its threads is stalled by a memory access.

High overhead memory access operations can be hidden by running thread sets alternatively. As the basic scheduling unit, the thread set has different names from different manufacturers. It is called a warp in Nvidia GPUs and a wavefront in AMD GPUs. The maximum number of thread sets is determined by the hardware resources,

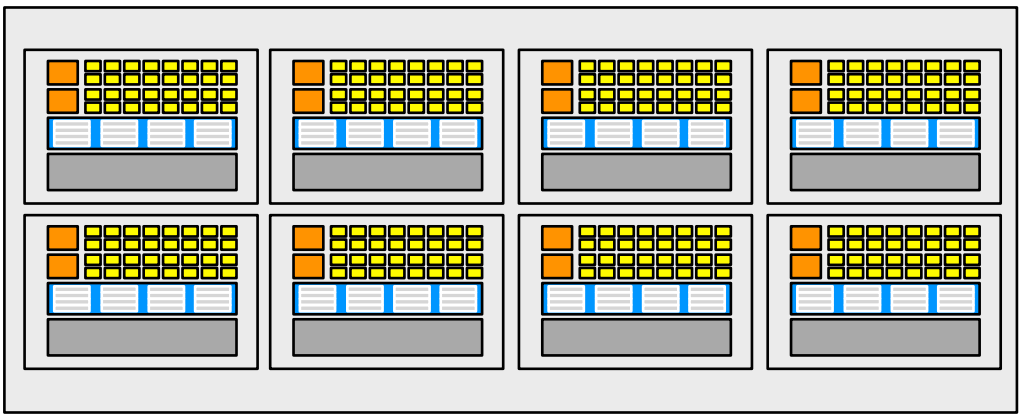


Figure 2.5: GPU with 8 SIMD processors. In each SIMD processors, there are 2 decoders and 32 computing units. Also, 4 context groups can reside in each SIMD processor simultaneously. Totally, 16 instructions can run in parallel at any given time, and 1024 contexts can reside in GPU to hide high-cost memory access. [Fatahalian (2008)]

e.g. the size of register files.

Multiple SIMD Processors

GPUs normally contain multiple SIMD processors. ALUs within each SIMD processor share the same local memory and therefore threads executing in the same SIMD processor can communicate with each other by sharing data. However, threads in different SIMD processors only share the same main memory, and the hardware does not support direct communication between them. The theoretical throughput of a GPU is determined by the number of SIMD processors. In figure 2.5, a GPU with 8 SIMD processors, each of which has 32 ALUs can run up to 256 threads at the same time. Typically, the number of resident threads is larger than the active threads so as to hide memory accessing latency. Therefore, a GPU can easily support thousands of concurrent threads execution. Typically, GPUs are accelerators that provide a massively parallel computation. It requires a new programming language to exploit its parallel structure.

2.2 OpenCL

Open Computing Language (OpenCL) is a framework for programming software applications that is portable across heterogeneous platforms. Since its initial release in

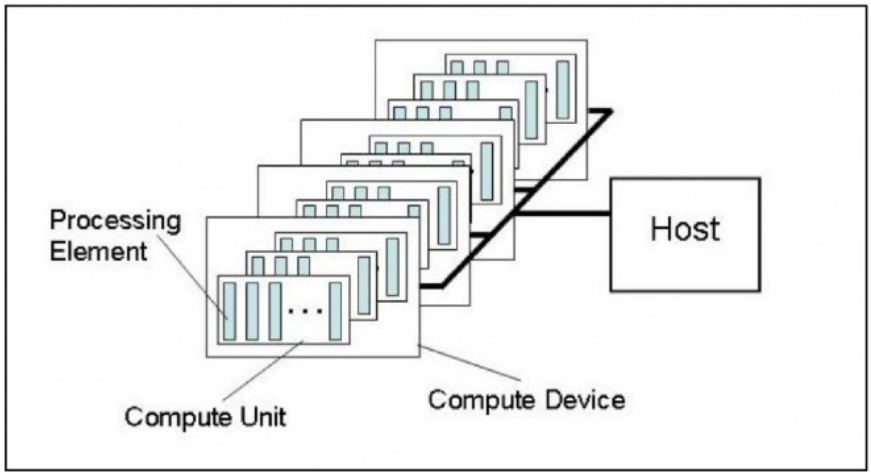


Figure 2.6: OpenCL platform model. One host device can manage one or more compute devices. Each compute device is composed of one or more compute units each with one or more processing elements. [Khronos (2016)]

2009, there are several mainstream processors which support the OpenCL standard. These include CPU, GPU, digital signal processors (DSP), field-programmable gate arrays (FPGA) and other hardware accelerators.

OpenCL specifies a programming language and a group of application programming interfaces (API) for programming the supported devices. The language is a C-like language that is extended from C99. For any given OpenCL program, it is divided into host code and kernel code. The host code carries out the primary control, such as data input/output and workload synchronization. It also maintains the working environment within which the kernel can access the selected device to perform its computation. The kernel code is portable across different devices. It is loaded and compiled at runtime. When it is executed, a large number of threads are created from the same kernel function but on different data.

2.2.1 System Model

The OpenCL system abstraction is shown in figure 2.6. Typically, it consists of a host device, and several compute devices. The host device must be a CPU processor while the compute devices can be any processor that supports OpenCL. In this thesis, the target compute devices include multi-core CPU and GPU.

The compute device consists of compute units which include processing elements. For CPU, it contains one compute unit which has several cores that work as processing elements. For GPU, the compute unit stands for the SIMD processor and the processing elements represents the ALUs in figure 2.2b. Typically, the host and compute devices are connected via PCI-Express. The code running on the host side manages the data movement and workload allocation.

There are three core concepts for the OpenCL system model, which are platform, device, and context. Each of them represents a level of hardware abstraction.

Platform The platform is a specific implementation of OpenCL that is provided by device vendors. It is an abstraction of various devices that are mainly manufactured by the same vendor. The platform abstraction hides the complexity of programming with the specific drivers and allows the high-level source code to be migrated to another processor by remapping to a new platform.

Device Devices are the physical processors that perform the computation. At the runtime, each device has a unique identifier (ID). Through the device ID, the program can manage data movement, loading the input data to the device and read back the result when the device has finished the computation.

Context The context represents a set of devices that are accessible to the program at runtime. It contains at least one device. When there are multiple devices in the same context, all the devices can be of different types. When the context is created, the programmer has to decide how many devices are to be contained by the context and provides the list of device IDs explicitly. Therefore, during the program's execution, all those devices are visible, but others are not.

Within the same context, the host and compute device can cooperate with each other. There is part of the typical work flow for every OpenCL program. The host and compute device communicate and synchronize with one another via a data structure named the command queue.

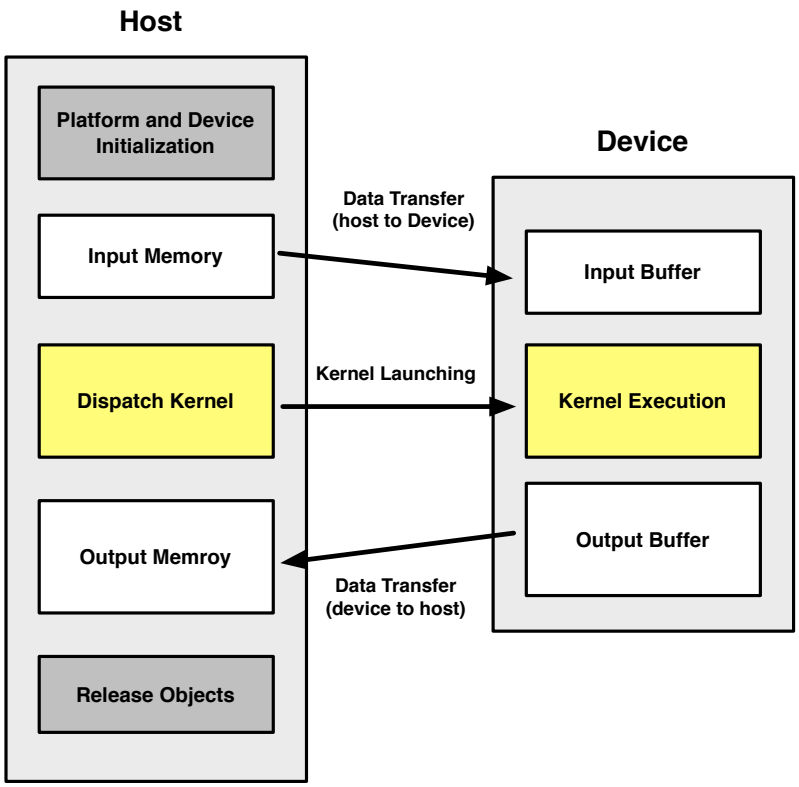


Figure 2.7: OpencCL workflow. The host manages platform oriented procedures while the device carries out the computation.

2.2.2 Work Flow

Every OpenCL program has a work flow similar to that is shown in figure 2.7. It contains several steps: context setup, data preparation, kernel deployment, result retrieve, and environment clean up. The host code follows this work flow by calling the corresponding OpenCL APIs.

Context Setup Every OpenCL program has to initialize the working environment by checking the platform and devices. A context is the created on the devices so as to manage data movement and kernel deployment among the devices. A command queue is a data structure that is created in the context. The host and device then can communicate via command queues. Normally, one command queue is bound to a device; however, in some cases, multiple command queues are bound to the same device for different reasons, such as maximizing bandwidth usage.

Data Preparation Once the context is setup, the host code creates buffers for the input data and then copies the data to the selected device. If the device is a GPU, for example, explicit data movement happens between the main memory and GPU memory. However, if the selected device is the host device itself, explicit data movement is not necessary. Instead of copying the data, pointers are passed to save cost.

Kernel Deployment The arguments of the kernel are set explicitly before its launch. The overall number of threads and the workgroup size have to be set as well. A detailed description about them is described in section 6.4.2. Then, the host launches the kernel to the target device via the bounded command queue. Kernel launching is asynchronous. The host code does not have to wait until the kernel finished; instead, it can carry on with the other following statements.

Result Retrieve The result retrieve is usually a blocking function which does not return until the data has been copied from the device. Similar to the data preparation, as long as the target device is the host itself, no explicit data movement is necessary.

Cleaning up Finally, before the program is finished, the environment has to be cleaned up. The runtime allocated objects, such as buffer objects and command queue, have to be released properly.

2.2.3 Kernel Execution Space

The launched kernel is instantiated as an object with a large number of threads, so as to run in parallel, especially on GPU processor. The thread instances (which are called work-items in OpenCL) are organized in an N-dimensional index space, namely NDRange, which is shown in figure 2.8. In OpenCL, the global size is used to describe the number of work-items in each dimension of the NDRange. For example, in figure 2.8, the global sizes on `x` and `y` dimension of NDRange are `Gx` and `Gy`.

The work-items in NDRange are further organized into work-groups. Each work-group contains the same number of work-items. In OpenCL, the size of the work-group is denoted by work-group size which describes the number of work-items in each dimension of the work-group. In the figure 2.8, the work-group size is represented by `(Wx, Wy)`. The work-group size is also called local-size.

Every work-item has a unique ID, called global ID, within the NDRange. Similarly, it has an ID within the work-group, called local ID. However, the local ID is unique

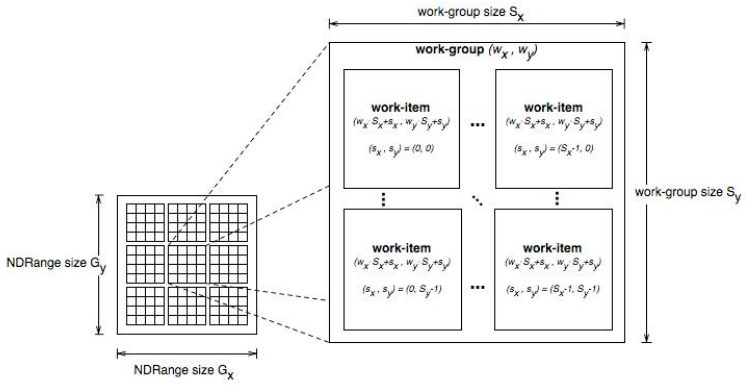


Figure 2.8: Two dimensional NDRange. A kernel is launched as multiple instances of threads each with the same kernel functionality. Each thread is a work-item in OpenCL. A fixed number of work-items are organized within a work-group and work-groups are organized within an N-dimensional range, namely NDRange. [Khronos (2016)]

within the work-group. Like work-items, work-groups also have unique IDs to identify themselves in the NDRange. The relationship of various objects in kernel execution space is shown in figure 2.8, and the index description is shown as follows:

- Size of NDRange: (Gx, Gy)
- Size of work-group: (Sx, Sy)
- Number of work-groups: (Wx, Wy)
- Work-item global ID: (gx, gy)
- Work-group global ID: (wx, wy)
- Work-item local ID: (lx, ly)

The following formulas correlate the work-item identifiers and the size of the execution space:

(gx, gy) = (wx*Sx+lx, wy*Sy+ly)

(Wx, Wy) = (Gx/Sx, Gy/Sy)

(wx, wy) = ((gx-sx)/Sx, (gy-sy)/Sy)

2.3 Machine Learning

Machine learning explores pattern in data. The learning algorithms build models from data which can be used in prediction. As a data-driven scheme, machine learning is different from traditional analytical models, which are derived by humans. The primary advantage of machine learning is that it can generalize models from historical samples to make an accurate decisions on unseen data in the future. Machine learning has been employed in a wide range of areas, such as Web searching, spam filtering, computer vision, and optical character recognition.

In this thesis, we use machine learning technique to predict OpenCL kernels device affinity and whether or not two kernels concurrent execution can provide a strengthened performance over running them sequentially.

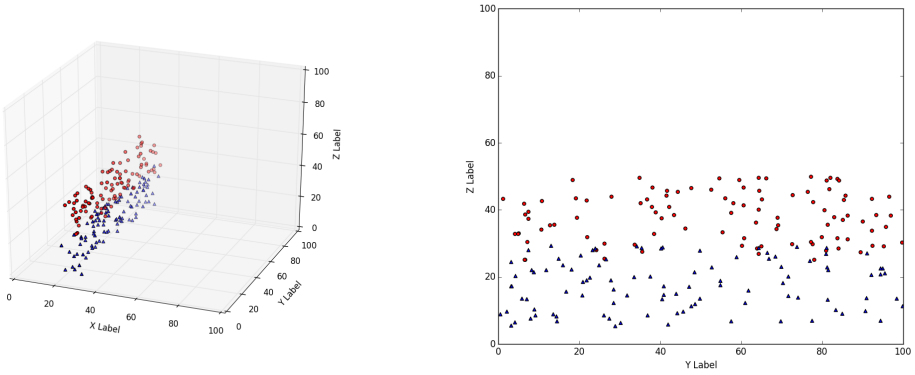
2.3.1 Feature-based Learning

Feature-based learning is a set of algorithms that learn from features, which are transformations of the original data. The features are a representation of the primary aspects of the input data and can be exploited by machine learning model effectively.

In feature based learning, an observation can be described by a pair $\langle \vec{x}, t \rangle$. Here the $\vec{x}$ represent a set of features, which is abstracted from the raw data, and t is the result, or target, that we are interested in. The goal of the learning is to generalize a model that discovers the relationship between features and targets from the training sample pairs.

Typically, feature based learning is divided into two categories: unsupervised learning and supervised learning. In unsupervised learning, the training samples only include the features, and no correct output targets are provided within the samples. In supervised learning, the algorithm learns from the training samples that contains both the features and the desired targets.

Unsupervised learning Unsupervised learning algorithms infer functions solely from the features to discover the input data similarity. As the input of the learning model, all features are treated equally. The goal of learning is to detect the regularities of the input and group the input data which are similar or have a close distance to each other.



(a) The original data in 3D space.

(b) After PCA the data space is reduced to 2D.

Figure 2.9: Example of using PCA reduce data set from 3D dimension to 2D dimension.

The canonical example of this method is clustering that partitions the observation inputs into several sets. In this thesis, the unsupervised learning approach is only used as a pre-processing operation to find out the Principal Component Analysis (PCA). A detailed discussion of PCA is given in section 2.3.2.

Supervised learning Supervised learning can be described as a set of observation pairs $\langle \vec{x}, t \rangle$. The goal of the algorithm is to build a model that can find out the relation between vector $\vec{x}$ and target `t`. In training feature pairs, target `t` is set explicitly to guide the learning; therefore, this learning procedure is supervised by the target. According to the values of target `t` are continuous or discrete, supervised learning is categorized into regression and classification. This thesis uses supervised learning to build a classifier to estimate the target kernels. Section 2.3.3 and 2.3.4 discuss the training algorithms that are used in this thesis.

2.3.2 Principle Component Analysis

PCA is a static method that converts a set of possibly correlated variables into a set of linearly uncorrelated variables via orthogonal transformation. The number of linearly uncorrelated variables, called principal components, are less than or equal to the original variables. Reducing the dimension reduces the complexity of the model, as the problem is simplified when it has fewer features. Some features convey very little information, e.g. they have small variances or can be expressed by other features linear combinations. The problem dimension can be safely reduced by removing these features without sacrificing model accuracy.

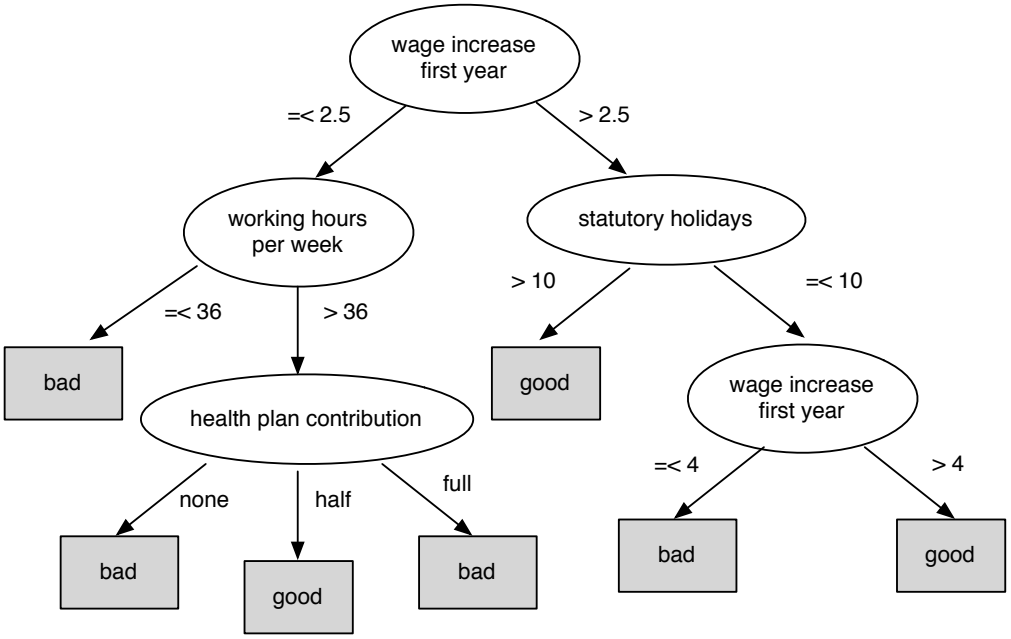


Figure 2.10: Decision Tree. A decision tree for labor negotiation: whether or not a contract will satisfy the employee. [Witten and Frank (1999)]

PCA can be thought of as projecting an n-dimensional ellipsoid to the axes. The principle components are organized in a way that the first component has the greatest variance by the projection of ellipsoid to the coordinate. This coordinate is called the first principle component. Then, it followed by the second greatest component, and so on.

Figure 2.9a shows an example of a set of data that has three components, or features. The variance of each component can be acquired by projecting the data to the coordinate that represents the component. The variances in y-axis and z-axis are very similar, but both of them are significantly greater than the variance along the x coordinate. Therefore, the component space can be transformed from three-dimensions to two-dimensions without losing much of the accuracy. The transformed feature graph is shown in figure 2.9b.

2.3.3 Decision Tree

Decision tree based classifier is a supervised learning technique that classifies an observation of an item to a category that concludes the item's target value. As the name suggests, the trained model is typically presented as a tree. The internal nodes of the tree test the features while the branches represent the results of the comparison.

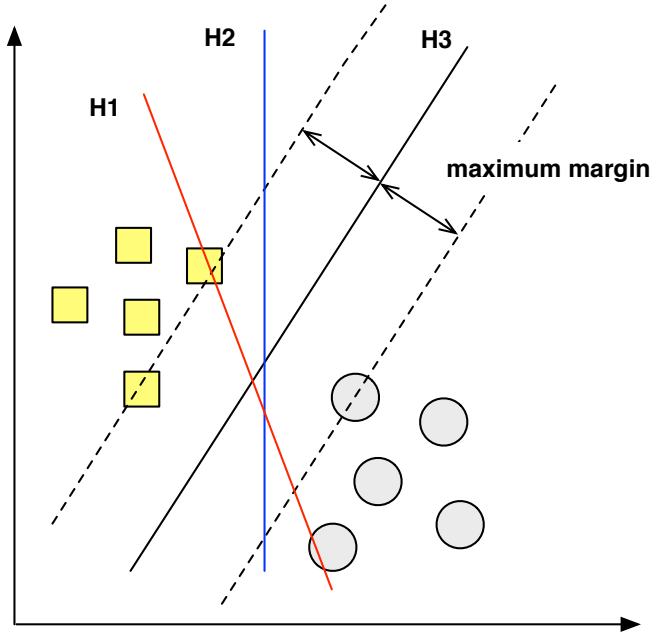


Figure 2.11: Separate data with hyperplane of the maximum margin. SVM algorithm attempts to find a hyperplane that can separate data nodes into two classes widely enough to each other. Though, both H1 and H2 can separate data nodes, their margins are small. H3 provides the best separation with the maximum margin.

The advantage of the decision tree is that it is simple to understand and interpretable. However, training an optimal decision tree is known to be an NP-complete problem. Compared to support vector machines the decision tree is often less accurate.

Decision trees are generally constructed in a top-down manner which includes two phases: growing and pruning. The greedy algorithm constructs decision tree recursively. In each iteration, the algorithm evaluates the partition of training and select the most appropriate split. After that, a further subdivision is carried out on the training set to split it into a smaller subset. The algorithm keeps splitting training data until no split gains it can have.

Though training a proper tree is involved, using it is simple. New data items are classified by traversing the tree from the root down to a leaf. At each internal node, the corresponding feature is tested. The outcome of the comparison guides the item to the particular branch and at that node, another comparison is carried out. Once the item reaches a leaf node, the classification is finished. The category of the leaf node is the data item's target class.

Figure 2.10 shows an example of a decision tree that classifies whether or not an

employment contract satisfies the employee [Witten and Frank (1999)]. If the first year wage increase is less than 2.5 and in every week the working time is less than 36 hours, it is not a good contract, as it ends up in the leftmost leaf node.

2.3.4 Support Vector Machine

Support Vector Machines (SVM) are a supervised learning technique. They can be used for classification. When used as a binary classifier, SVMs determine whether a data item belongs to one class or the other. A data item with n features can be viewed as an n-dimensional vector. The function of SVMs is trying to find out an (n - 1)-dimensional hyperplane to separate the data items. For a number of data items, multiple hyperplanes existed that can partition them. The reasonable choice of the best hyperplane requires that the data items are widest separated. Therefore, the function of the linear SVMs classifier is to learn from the data to find the hyperplane that has the maximum-margin.

Figure 2.11 shows and example of three hyperplanes. Only the H3 separates the data items with the maximum margin. Therefore, the function of H3 is the model SVM has learned from those training data. For new points, they will be classified to one of the categories if their coordinate is on the top left of H3. Otherwise, they will be categorized into the other class.

2.4 Benchmarks

Table 4.3 lists the OpenCL programs that are used in this thesis. We selected kernels from a wide range of benchmark suites, which include Nvidia SDK, AMD SDK, Parboil, and polybench. The exact configurations, benchmark subset selection, and input setting are described in the experiment setup sections of following chapters.

2.5 Evaluation Method

Performance Metrics: To evaluate our approach, we used two metrics, *system throughput*, a system oriented metric, and *turnaround time*, a user oriented metric. Those two metrics have widely been used to evaluate the performance of a scheduler in a multi-tasking environment [Snavely and Tullsen (2000); Eyerman and Eeckhout (2010)].

Table 2.1: Benchmarks

Suite	Benchmarks	Benchmark
NVIDIA	BlackScholes	ConvolutionSeparable
	DXTCompression	DotProduct
	FDTD3d	HiddenMarkovModel
	Histogram	MatrixMul
Parboil	BFS	Cutcp
	Sgemm	Spmv
AMD	BinarySearch	BinomialOption
	BitonicSort	BlackScholes
	BlackScholesDP	DCT
	DwtHaar1D	FastWalshTransform
	FloydWarshall	Histogram
	MatrixMultiplication	MatrixTranspose
	PrefixSum	QuasiRandomSequence
	Reduction	ScanLargeArrays
	SimpleConvolution	
Ploybench	ATAX	BICG
	CORRELATION	GESUMMV
	SYR2K	SYRK
	2DCONV	3DCONV
	GEMM	GRAMSCHMIDT
	2MM	3MM
	COVAR	FDTD-2D
	MVT	

Our goal is to maximize the system throughput which in general leads to favourable turnaround time results. The definitions of the two metrics are given as below.

System throughput (STP) is a *higher is better* metric. It describes the number of tasks completed per unit time. This is calculated by using the baseline of 1.0, showing the relative speedup of other scheduling policies. It is defined as

$$STP = \frac{\sum T^i_{Baseline}}{\max(\sum T^m_{cpu}, \sum T^n_{gpu})} \tag{2.1}$$

where $T^i_{Baseline}$ is the execution time given by baseline, and T^i_{cpu} and T^i_{gpu} are the execution time by running task T^i on the CPU and the GPU respectively.

Average normalized turnaround time (ANTT) is a *smaller is better* performance metric. It quantifies the time between a task is created and its completion, indicating the average user-perceived delay in multi-tasking environment compared to running a

single task on the system. In the experiments, the turnaround time is normalized to the baseline scheme. ANTT is defined as:

$$ANTT = \frac{1}{n}\sum_{i=1}^{n}\frac{T^i_{sch}}{T^i_{Baseline}} \tag{2.2}$$

where $T^i_{Baseline}$ and T^i_{sch} are the time between task T^i is created and its completion using baseline and an alternative scheduling policy respectively.

2.6 Summary

This chapter presents an introduction to heterogeneous systems. CPU-GPU systems have been widely used in recent years and are the target of this thesis. Then OpenCL standard as the most significant framework is introduced. Applications can be programmed once and performed across different processors. The framework provides a unified programming interface to hide the difference between various processors. A brief introduction of machine learning is presented in the third section of this chapter. Finally, this chapter presents the evaluation method and the benchmarks used in this thesis.

Chapter 3

Related Work

This chapter discusses related work in task scheduling for heterogeneous systems. Section 3.1 discusses single application scheduling across multiple processors. This work uses workload partitioning to improve single program performance and power efficiency. Section 3.2 presents work on multiple tasks space sharing on the same processor. Such a scheduling method aims to improve hardware utilization via high job concurrency. Section 3.3 discusses work on scheduling multiple tasks to multiple devices. The goal of this scheduling is either to improve overall system throughput or to decrease power consumption. Section 3.4 describes work that tries to minimize data communication overhead. Finally, in section 3.5 and section 3.6, two different modeling approaches are discussed: analytic modeling and machine learning.

3.1 Scheduling Single Task to Multiple Devices

This section describes prior work in scheduling a single task to multiple devices on heterogeneous platforms. There are two different platforms considered in this section: single-ISA and separate-ISA system. In the single-ISA system, single job scheduling involves multiple threads split across separate devices and task migration. For separate-ISA platform, the primary approach is task partitioning, which is used to improve performance and balance workload.

3.1.1 Single-ISA Heterogeneous Platform

Scheduling for single-ISA platforms is either implemented dynamically by the runtime or operating system, or statically by the compiler, by mapping threads to different

physical cores.

Lakshminarayana et al. (2009) propose a runtime approach called Age-Based Longest Job Fast Core First scheduling policy for asymmetric multiprocessors system. The proposed scheduler predicts the remaining execution time of threads and assigns threads with a larger estimated remaining time to a fast core. The execution time prediction is based on the thread's relative age, based on the observation that threads which are created around the same time tend to have similar execution times. It schedules the thread, which is created with other threads at the same time but has lagged behind, to the fastest core in order to improve performance.

Li et al. (2007) present an operating system scheduler which works on both SMP and NUMA-systems. The scheduler extends the conventional operating system load balancing scheme. The proposed approach maintains the workload on each core proportional to its computing power. A NUMA-aware migration controller optimizes thread migration by predicting its migration overhead. The scheduler adopts a Faster-Core-First algorithm. However, the scheduler extended the conventional method by allowing threads to migrate to cores that have lower scaled load even when some of their original cores may become idle. This migration ensures threads always run on faster cores whenever they are under-utilized.

Shelepov et al. (2009) propose a compiler approach: Heterogeneous-Aware Signature-Supported scheduling algorithm (HASS). HASS is a static method that makes the program-core mapping based on the thread's architecture signatures. The thread's architectural signature is collected offline, which includes a set of properties, such as memory-boundedness, available ILP, sensitivity to variations in clock speed and other parameters. At runtime, the scheduler analyzes a thread's offline profiled signatures to estimate its performance on each type of core to find out the best match.

Though single program mapping to separate single-ISA processors works well in improving throughput and energy consumption, this approach cannot be used on heterogeneous platforms that have processors with different architectures and instruction sets. For single-ISA cores, their differences are in the micro-architecture, i.e. frequencies and cache sizes. Migrating threads from one core to another is straightforward, as no code recompilation is required. As many single-ISA platforms share memory or last level cache (e.g. ARM bigLittle [ARM (2016)]), communication overhead is much smaller than in separate-ISA systems.

3.1.2 Separate-ISA Heterogeneous Platform

Partitioning also is the core approach to scheduling a single job on multiple separate-ISA processors. OpenCL is the dominant standard for cross-platform programming and there has been much research in kernel partitioning. The core idea in kernel partitioning is to split a kernel into parts, each of which will be allocated to a distinct device. The partitioning could be performed either staticly or dynamically and different approaches focus on achieving the best deployment to balancing kernel slices across various devices.

Zhong et al. (2012) propose a model for scientific applications, such as linear algebra routines, to execute them efficiently on CPU-GPU heterogeneous systems. The target system is a hybrid platform that consists of CPU and GPU; each comes with a dedicated memory system. The functional performance model (FPM) represents processor speed as a function of problem size and a set of significant features that characterize the architecture and application. Based on the model, a data partitioning algorithm takes place on the target application to improve its performance. The workload aims to be balanced on the platform, as the application is partitioned across multicore CPU and multi-GPU proportionally.

Grewe et al. (2013) propose a compiler-based approach that generates optimized OpenCL code from OpenMP programs. A machine learning adaptive model is developed in this paper to predict whether to perform the original OpenMP application on the multicore CPU or generate optimized OpenCL kernel and partition this kernel across CPU and GPU.

Ghose et al. (2016) present an in-depth analysis of control flow divergence of OpenCL kernels. Since branches have a significant impact on OperCL kernel performance, in this paper, the author uses divergence as a guide to partition a kernel across CPU and GPU. A machine learning model is trained by using the amount of divergence in a program; then this model is used to predict unseen program's optimal partition.

Kaleem et al. (2014) propose an online profiling-based approach to partition application across CPU and GPU. The scheduler works in two phases: profiling and execution. First, the given kernel executes a subset of work items on both CPU and GPU. By profiling the workload on CPU and measuring the execution time on GPU, the scheduler calculates the partition ratio for the remaining kernel work items.

Pandit and Govindarajan (2014) present a runtime framework, namely FluidiCL, to run a single OpenCL application on both CPU and GPU. FluidiCL divides the application into a set of small fractions, called sub-kernels. Each sub-kernel works on several workgroups. The NDRange of the original kernel is flattened to a one-dimensional NDRange so as to enable CPU and GPU taking workgroups from either end of the new NDRange. The host CPU controls data transfer from the main memory to the GPU buffer and controling the status for each workgroup. The GPU kernel periodically checks the status set by the CPU in the first work-item of every workgroup. If it finds the workgroup has already been completed by the CPU, it exits the current kernel and transfers the local results back to the CPU main memory. This is followed by result merging. We have compared our work with FluidiCL in Chapter 4.

Kernel partitioning is an effective method to speed up single kernel's execution time by running separate kernel slices on different devices concurrently. It is also a promising approach to balancing the workload across processors. However, greedily speeding up every single kernel does not necessary lead to an optimized overall performance when multiple applications are queueing for the processors. Therefore, a different approach is required for targeting multi-tasks scheduling.

3.2 Scheduling Multiple Tasks to Single Device

Scheduling multiple tasks to a Single-ISA core is normally managed via time sharing. This is a well-known technique and not discussed here. This section discusses the work of multiple task scheduling on Separate-ISA heterogeneous platform. The most popular idea among the following work is to space share the accelerator by multiple workloads.

3.2.1 Separate-ISA Heterogeneous Platform

Scheduling multiple tasks to a single device has been investigated for CPU-GPU platforms. Researchers note that GPU hardware resources are often underutilized. A kernel from one application may consume one kind of hardware resource heavily and leave others underutilized. Therefore, multiple tasks can share the GPU if they are bounded by different kinds of resources. Concurrent kernel execution, or kernel fusion, is the method that improves system throughput and GPU hardware utilization by running multiple kernels on the same GPU simultaneously.

Zhong and He (2014) proposed a runtime system, namely Kernelet, to support concurrent execution of CUDA kernels on a single GPU to improve its throughput. Kernelet slices the kernel into small pieces and then co-schedules it with other slices from different kernels. Each kernel slice contains several CUDA blocks. The sizes of the kernel slices are determined by Kernelet at runtime. Kernelet provides a performance model for the purpose of selecting a pair of kernels for co-scheduling and deciding the slice sizes for each of these kernels. After the target kernels are selected, the scheduler deploys the kernels pair by pair using a greedy algorithm. The GPU is space-shared by these kernels based on the ratio of their slice size.

Wang et al. (2010) propose a kernel fusion method to improve GPU utilization and power efficiency. Based on CUDA, three kernel fusion approaches are proposed in this paper, which are Inner Thread Kernel Fusion, Inner Thread Block Kernel Fusion, and Inter Thread Block Kernel Fusion. By adopting dynamic voltage and frequency scaling (DVFS) to the fused kernel, the presented method aims to optimize energy consumption.

Wang et al. (2011) propose a runtime framework, namely context funneling, to make CUDA kernels from different applications run concurrently. Targeting Fermi architecture, the runtime framework provides a virtual context that could be shared by all CUDA applications. Therefore, kernels from separate applications could run concurrently via distinct CUDA streams under the same context.

Adriaens et al. (2012) provide a spatial multitasking method for the GPU to improve resource utilization. Profiling information is used to choose the best partition of CUDA stream machines. Based on profile information of all candidate programs, the heuristic finds the best partition of GPU hardware and schedules a sub-set of application to space share the GPU.

Tarakji et al. (2015) proposed a GPU scheduler to improve hardware resource utilization. The scheduler in this paper works on the NVIDIA Kepler architecture GPU. Given a group of applications, the scheduler splits their grids associated with the corresponding kernels into slices. Then, instead of launching each kernel as a whole, the scheduler makes multiple kernels share the device to improve resource utilization.

Wende et al. (2012) propose a producer-consumer based runtime approach to manage concurrent kernel execution on a single GPU device. Targeting host programs that have multiple threads running in parallel, where multiple kernels could be issued. This paper presents a runtime system that uses a producer-consumer model to solve the synchronization problem. The host thread place kernels onto different CUDA streams. All

the kernels are placed into a queue, from which the consumer dequeues the kernel and issues it to the GPU.

Awatramani et al. (2013) propose a block scheduling method to support concurrent kernel execution. A new hardware block scheduler is described in this paper. The core component of the block scheduler is a kernel status table that keeps track of all arrive kernels. When issuing blocks, the scheduler checks the valid bit of each kernel and launches them alternative to the stream machine by a round robin algorithm.

Aguilera et al. (2014c) present a kernel concurrent execution approach that takes quality-of-service (QoS) into consideration. The method assumes at least one application has a requirement on QoS. The proposed framework first checks the application and predicts its minimum resource requirement; Then, it dynamically allocates the number of stream machines to the application and runs kernels from other kernels that are not sensitive to QoS on the rest of stream machines.

Aguilera et al. (2014a) present a spatial multi-tasking approach of within-die (WID) GPU that shares the same die with CPU. This allows more stream machines integrated into the same die and working under different clock frequencies. This paper proposes a variation-aware workload partitioning approach that decides how many hardware resources to assign to each application by characterizing the sensitivity of each application's performance to the stream machines.

Lee et al. (2014) present a hardware solution to support concurrent kernel execution. Two separate aspects of thread block scheduling methods are examined in this paper, which is Lazy CTA scheduling (LCS) and block CTA scheduling (BCS). Among these two scheduling aspects, LCS restricts the maximum number of blocks that could be scheduled to the stream machine and BCS controls how consecutive blocks are issued to the same stream machine.

Aguilera et al. (2014b) examines various resource sharing policies that impact performance and fairness. The observation is that the resource allocation that is optimized for the performance is not necessarily best for fairness. This paper proposes a metric to measure the fairness of resource sharing. A runtime approach is proposed that comes with several scheduling policies, such as equal compute resources for each application, equal throughput for each application, equal speedup for each application, and maximum system throughput. A runtime algorithm is then implemented to adjust stream machine allocation among applications according to their predicted resources requirement under each policy.

Guerreiro et al. (2015) present an approach to tune the performance and energy consumption for concurrent kernel execution. By profiling kernels, the auto-tuning procedures explore the single- and multi- kernel's performance and energy optimization space, configure the thread block and set the GPU operating frequency accordingly.

Although concurrent kernel execution improves GPU utilization and thereby optimizes throughput, it does not take the multi-core CPU into consideration. In practice, many applications have better performance on multi-core CPUs than on the GPUs, particularly when taking data movement overhead into consideration. Also, since all kernels execute on GPU, there is an imbalance between CPU and GPU. The key limitation of the above approaches it that they do not utilize the multi-core CPU.

3.3 Scheduling Multiple Tasks to Multiple Devices

When scheduling multiple tasks to multiple devices, the primary function of the scheduler is to find out the most appropriate device for each application. In the following sections, we describe prior work in both single-ISA and separate-ISA systems.

3.3.1 Single-ISA Heterogeneous Platform

Programs on single-ISA platforms typically have a better performance on the faster core, but this performance comes at a cost of greater power consumption. The task scheduler on single-ISA system tries to find the balance between maintaining the overall throughput and limiting the energy consumption at the same time. One common method adopted by the single-ISA system is to make the scheduler deploy programs to their best-fit device, with the help of compiler and architecture model, to meet a power and performance budget.

Chen and John (2009) present a framework to leverage the relationship between program characteristics and hardware resource requirement for program scheduling in heterogeneous multicore processors. In this method, the program's estimated resource demand and the core's physical configuration are projected to a unified multi-dimensional space. By using weighted Euclidean distance in this space, the best program-core matching can be found and used as a guide in program scheduling.

Muck et al. (2015) provide an accurate runtime model to estimate performance and power consumption. Implemented on top of the Linux kernel, the runtime model

(Run-DMC) uses a fine grain per-thread metric to model instruction level parallelism, thread load contribution, and scheduling policies to predict the power and performance across all core types in the system. Run-DMC allows the program to find its most appropriate core type when fully exploiting the heterogeneous platform. In practice, by employing an evolutionary optimization algorithm, Run-DMC can find good thread-to-core mappings to maximize overall energy efficiency.

Koufaty et al. (2010) propose a bias-based scheduler for heterogeneous system with single-ISA cores. Since different applications behave differently on different cores, the scheduler monitors the application bias then selects the most appropriate core for the application. Here, the bias is defined as the type of core that application would prefer to run at a particular time. Usually, a thread has a small core bias if its speedup brought by running it on a larger core would be modest. On the other hand, a thread has a large core bias if its speedup from executing on a larger core is significant compared to running it on a small core. Performance counters are used to keep track of each thread's states, which are essential for thread bias computation. Then, the scheduler deploys programs to the cores according to their bias.

Craeynest et al. (2012) develops a Performance Impact Estimation (PIE) mechanism to schedule workload to single-ISA heterogeneous platforms. PIE is a model that uses profiling information, such as CPI (Cycle Per Instruction) stack, MLP (Memory Level Parallelism) and ILP (Instruction Level Parallelism), to estimate workload performance on different core types. According to the estimated performance, PIE dynamically adjusts the scheduling scheme at the runtime to optimize the program mapping and thereby improve the performance and energy consumption.

Becchi and Crowley (2006) propose a dynamic thread assignment for single-ISA heterogeneous platforms. Two dynamic assignment approaches are provided, which are Round Robin Dynamic Assignment and IPC-Driven Dynamic Assignment. The round robin scheme is blind to thread runtime behavior. However, it is the simplest strategy in practice and brings an improved performance compare to the static assignment. The IPC-Driven assignment takes thread characteristics into consideration. It uses the average of IPC as the control variable when assigning threads to physical cores. The controlling variable is computed at runtime and used as a guide to trigger thread migration between cores.

Though the above approaches are effective on the single-ISA platform, they cannot directly be used on separate-ISA systems, as scheduling decisions are based on estimating the same instruction stream on separate hardware configurations. On separate-ISA

platforms, programs are compiled and optimized for different processors separately; and for each type of device, a dedicated power and performance model is needed for the best program-device pair selection. Migrating jobs between devices can also lead to a considerable runtime overhead introduced by recompilation and data movement. Therefore, for separate-ISA heterogeneous systems, different approaches have been tried.

3.3.2 Separate-ISA Heterogeneous Platform

Most separate-ISA scheduling work is targeted at on CPU-GPU system. In this section, the main idea is to find the most appropriate device for a program via runtime metrics and predictive models.

Lang and Rünger (2014) propose an execution time and energy model for parallel Conjugate Gradient Method (CGM) on a CPU-GPU heterogeneous system. The model is developed based on the platform information, such as the speedup of CPU and GPU, their voltage and frequency scaling, the energy consumption of the memory and the energy consumed by moving data between CPU main memory and GPU memory. On a given machine, the model makes it possible to exploit the most energy-efficient distribution of workload between multicore CPU and GPU.

Ravi et al. (2012) studies the problem of optimizing the overall throughput of a group of applications on a heterogeneous platform that consists of multi-CPUs and many-core GPUs. Two different scheduling problems are considered in this paper. In the first one, the jobs can be executed on a single node, either CPU or GPU. In the second one, the jobs can be served by CPU, GPU, or both. For the first problem, three scheduling heuristics are developed, namely Relative Speedup based policy with Aggressive Assignment (RSA), Relative Speedup based policy with Conservative assignment (RSC), and Adaptive Shortest Job First policy (ASJF). Different schemes expect a different amount of information from the users, but they all outperform the default round robin method. For the second problem, a Flexible Moldable Scheduling method (FMS) are proposed to improve the throughput by modeling the resource type and the number of requested nodes.

Barak et al. (2010) present a package that enables OpenMP, C++ and unmodified OpenCL applications running on clusters with many GPU devices. This Many GPU Package (MGP) includes the implementation of OpenCL and OpenMP extension that allow applications to use various devices (CPUs, GPUs, or both) transparently. It

also provides MOSIX-like algorithm for dynamic resource management to balance the workload then make the resource fairly shared by all applications.

Augonnet et al. (2010) extended StarPU by taking data transfer overhead into consideration. An asynchronous data prefetching method is proposed in this paper together with a data-aware scheduling policy. The proposed method not only exploits the benefit of asynchronous data transfer but also saves memory transfers by introducing scheduling policy to exploit data locality.

Many of the above approaches use program profiling information to estimate the best device. Profiling ahead is an effective method, however, in practice, it introduces substantial runtime cost that outweighs its benefits. When scheduling unknown programs at runtime, these approaches cannot be used.

3.4 Communication Optimization

Data movement and host-to-device communication may incur large runtime overhead for heterogeneous systems, particular is for those platforms that have discrete memory systems. In this section, we review some of the prior work on optimizing data movement.

Dathathri et al. (2013) propose an automatic data movement method to optimize the communication between computing devices in heterogeneous platforms with separate memory systems. This approach targets nested loops with affine bounds and accesses. By using static analysis and lightweight runtime routines, the compiler framework carries out a source-to-source transformation that generates the communication code. Then, by partitioning data dependencies, the approach proposed in this paper statically optimizes the data transferred between devices.

Jablin et al. (2012) present a Dynamically Managed Data (DyManD) system that works on automatic CPU-GPU data movement optimization. DyManD consists of a runtime library and a set of compiler passes. The compiler inserts the memory allocation calls to the original program and generates DyManD compliant code for the GPU. The runtime library manages data and optimizes communication. Similar to CGCM, DyManD adopts an acyclic CPU-GPU communication pattern to keep the latency off the program's critical path and allow parallel execution between CPU and GPU.

Jablin et al. (2011) present communication optimization methods for CPU-GPU heterogeneous platform. The CPU-GPU Communication Manager (CGCM) uses compiler modification and runtime library to manipulate data movement between CPU and

GPU, so as to effectively reduce data transfer overhead.

Optimization of data movement and overlapping data transfer and computing workload improves system performance. However, these methods offer no help in finding programs their best-fit devices and the relative executing orders. Also, similar to the problem in single task scheduling, communication optimization cannot improve hardware device utilization.

3.5 Analytical Models in Heterogeneous Scheduling

Many approaches use a performance model to estimate the performance of applications that run on the system. Analytical models typically use a set of equations to characterize devices and describe the action of a program on the target device. Information about the hardware configuration and program characteristics are needed for the model equations. In practice, the hardware configuration is modeled by running a group of micro-benchmarks to test the behaviors of the platform. Applications characteristics, which include memory footprint, the amount of instructions of each type, registers usage, the number of branches, are acquired by source code analysis. Analytical model can be classified into several groups that estimate program execution time, application bottleneck, power consumption and program behavior.

Execution Time Prediction

Hong and Kim (2009) models CUDA warp behavior by two metrics: MWP and CWP, in which the first one measures the number of warps that can overlap the memory access latency and the second one measures the number of warps that could execute their instructions in other warps memory accessing period. Based on the warp parallelism, the model could estimate a program's execution time according to its cost on memory request. Due to its high accuracy, many other models adopt it in their performance analysis.

Kothapalli et al. (2009) propose two models to predict the performance of CUDA kernels. The MAX model uses the max value between computing and memory accessing cycles. The SUM model uses the sum of the cycles of computing and memory accessing.

Kerr et al. (2010) uses statistical analysis to derive the relationship between program behavior and heterogeneous processors. This is then fed into the model to predict the performance of programs that have similar behavior on different processors.

Meng and Skadron (2009) specifically models the Iterative Stencil Loops (ISL)

application. The framework proposed in this work automatically selects the best parameters for ISL to improves its performance.

Choi et al. (2010) present an auto-tuned implementation of Sparse Matrix-Vector (SpMV) multiplication. The model is built for finding the best application parameters that minimize the execution time.

3.5.1 Application Bottleneck Analysis

Williams et al. (2009) provide a mechanism, namely the Roofline model, to describe the characteristics of a multi-core platform that may limit the performance of a particular application. Jia et al. (2012a) extended Roofline to guide kernel optimizations on GPU. It shows how features that affect performance on GPU by pointing out the bottleneck of the kernels.

Lai and Seznec (2012), Zhang and Owens (2011) and Baghsorkhi et al. (2010) provide a breakdown of a CUDA kernel execution into several stages. Platform information is gathered via micro-benchmark and the application is characterized by code analysis. Baghsorkhi et al. (2012) present a memory hierarchy model for GPU to predict the efficiency of the memory system. Sim et al. (2012) present an analytical model for GPU architectures and a framework that suggests the most profitable combination of optimizations for a CUDA kernel to improve its performance.

Zhang and Owens (2011) present a microbenchmark-based performance analysis model to identify the performance bottlenecks and guide programmers for optimization. The performance model uses microbenchmark to test three primary GPU components, which are instruction pipeline, shared memory, and global memory. Then, for general programs, the model uses the number of instructions, shared memory, and global memory transactions to detect the bottleneck.

Cui et al. (2012) present a model to estimate the performance of computation-bound GPU kernels with control flow divergence. Widely used metrics, such as divergent branches and divergent warp ratios, represent the divergence problem but do not indicate the impact on performance, in this paper, a new metric is proposed to model kernel performance. The metric is based on the basic block vector (BBV), instruction-based weighting, and thread-block level scheduling. The metric is used to estimate kernel performance, which has been further used as a value function for creating thread regroupings.

Karami et al. (2013) propose a statistical performance model for OpenCL applica-

tions. By using kernel performance parameters, a regression model is developed. The bottleneck of the kernel is analyzed by using PCA technique. The trained model can also leverage the unknown applications based on their performance similarities with the existing programs in the training database.

van Werkhoven et al. (2014) present a performance model that includes data transfer and computation and communication overlap for CPU-GPU systems. As data movement via PCI-E has a significant impact on performance, it can limit performance on CPU-GPU platforms, such as overlapping kernel computation and data communication. With the help of the PCI-E model, different methods for data transferring and computation-communication overlapping could be examined to select the best.

3.5.2 Power Consumption Model

Wang and Ranganathan (2011) and Hong and Kim (2010) propose an analytical power model. Both require the execution of micro-benchmarks and external power consumption meters to characterize the target devices. GPUWattch by Leng et al. (2013) and GPUSimPow by Lucas et al. (2013) present power model that work with GPGPU-Sim. Both of them are adaptations to GPUs of the McPat power modeling for multi-core architecture (Li et al. (2009)).

3.5.3 Simulations

Two popular simulators for GPU accelerator are GPGPU-Sim (Bakhoda et al. (2009)) and Barra (Collange et al. (2010)). Both are functional simulators of NVIDIA GPUs. A collection of user-configurable parameters is required to fine-tune the target device it simulated.

Apart from GPGPU-Sim and Barra, some other simulators simulating CPU+GPU, such as MacSim [HPArch (2012)], FusionSim [Zakharenko et al. (2013)], and Multi2Sim [Ubal et al. (2012)].

By modeling different types of computing instructions, cache and global memory access, bandwidth, and data transferring overhead, the model effectively identifies the bottleneck of a program, the best partition of a task, and even task execution time on a particular device. However, program profiling information is necessary when using the model in estimation, which limits the usage of this method in practice. Scalability is another problem for the model based approach. A new device added usually means a corresponding model should be built accordingly. A more flexible approach is to use

machine learning-based methods, which will be discussed in the following section.

3.6 Machine Learning in Heterogeneous Scheduling

Machine learning techniques have become increasingly important in task scheduling on heterogeneous systems. Though various performance models have been built to analyze applications performance then guide the task to select the most appropriate device, these models are hard wired by the authors and cannot adapt to new programs and hardware platforms automatically. Machine learning based methods instead learn from the data that are collected from programs and hardware configurations.

Machine learning has been used in power and performance analysis. In some work (Nagasaka et al. (2010), Song et al. (2013)), samples of performance counters and energy measurements at given frequency are used to train a model, which is used to identify power and performance bottleneck for the applications.

Bogdanski et al. (2011) uses machine learning to choose parameters for task scheduling and load balancing on a heterogeneous with GPU and FPGA. Paragon, proposed by Delimitrou and Kozyrakis (2013) uses collaborative filtering techniques to classify an unknown incoming workload on heterogeneity. The classification allows Paragon to schedule applications in a manner that minimizes interference and maximizes server utilization.

Song et al. (2013) uses an artificial neural network to estimate GPU power consumption and identify power-performance bottlenecks. The trained model is used for studying the energy efficiency for a single GPU-based system and energy-performance efficiency for GPU cluster. Dao et al. (2015) present a machine learning-based model to estimate the runtime of an arbitrary OpenCL kernel. With the information of cache usage and branch divergence, the machine learning model detects the relationship between these factors and the kernel's execution time.

Jia et al. (2012b) propose a regression-based performance model for GPU design space exploration. Stargazer sparsely and randomly samples parameter values from a full GPU design space, then uses these samples to train a machine learning-based estimator to predict unseen kernel's performance.

Kerr et al. (2010) uses a machine learning based model to predict the performance of CUDA applications on CPUs and GPUs. The static features of the program are used to derive the relationship between the program behavior and the performance on the target architecture. Luk et al. (2009) proposes a linear regression-based model to

distribute workloads to a CPU-GPU heterogeneous system. Unlike the above works, the linear model in this work needs to be trained for each newly encountered kernel.

Grewe et al. (2013) uses machine learning to predict the best partition of an OpenCL kernel between CPU and GPU. This static partition gets rid of the runtime overhead of profiling every application to split the task in a proper way. The feasibility of such an approach is guaranteed by the offline trained machine model. Jiao et al. (2015) introduces neural networks based machine learning model to find out the best pair of CUDA kernel that could improve the system performance by running in parallel.

As a very promising approach in building up the relationship between applications and target processors, machine learning is now widely used. However, the analytical models built by machine model focus on identifying the bottleneck of the application and/or the system. It could guide the optimization in detail, but has few contributions in runtime scheduling. Machine learning based task partitioning is an effective approach in balancing single task on CPU and GPU, however, it has not been used for multiple tasks from separate users. Concurrent kernel execution using machine learning techniques can help avoid the overhead introduced by a wrong kernel combination, but such an approach does not take CPU into consideration, which introduces an inter-processor workload imbalance. Unlike the above work, this thesis takes both separate and concurrent kernels scheduling into account using machine learning.

3.7 Summary

This chapter presents a brief review of prior work related to this thesis. These works cover task scheduling on heterogeneous systems, communication optimization, performance and energy models, and machine learning based runtime optimization.

In the next chapters, we present our smart runtime scheduler for CPU-GPU heterogeneous systems. By selecting OpenCL kernels their most appropriate device and best concurrent kernel pair, our scheduler improves the system performance and hardware utilization.

Chapter 4

Multi-Task Scheduling

This chapter examines multi-task scheduling on CPU-GPU heterogeneous platforms. Our approach extracts static features of OpenCL kernel and dynamic parameters, which are fed to an offline pre-trained machine learning based model that predicts whether or not it will have a high speedup when running on GPU compared to executing on CPU. Based on the estimated speedup, the scheduler inserts the candidate kernels into a double-end sorted queue. From either end, tasks are dynamically dequeued and executed on the appropriate device.

This chapter is organized as follows: Section 4.1 introduces the task scheduling on the CPU-GPU system. It is followed by a background section on OpenCL task scheduling in Section 4.2. In above sections, we present some of the background techniques and the most related works which have been discussed in Section 2 and 3, so as to highlight their relevance to this chapter. Section 4.3 provides a motivation example that shows that by scheduling kernel to the appropriate device we can optimize system throughputs. Section 4.4 describes the overall approach presented in this chapter. Section 4.5 describes the machine learning based predictive model construction. Section 4.6 gives more details on how we schedule OpenCL kernels at runtime. In Section 4.7, multiple alternative scheduling policies are considered. Section 4.8 and 4.9 describe how the experiments were carried out and the results of the experiment. Finally, there is a detailed analysis of our approach in Section 4.10, which is followed by the summary of this chapter.

4.1 Introduction

We now live in the parallel manycore era. Due to power-density constraints, increased single processor performance via ever-increasing clock frequency is no longer possible. This move to parallel system has been mirrored by the growing use of specialised accelerators such as GPUs. Heterogeneous systems consisting of multiple CPUs and GPUs are increasingly attractive as they provide cost-effective, energy-efficient high performance computing

OpenCL [Khronos (2016)] has emerged as a standard which provides program portability by allowing the same program to execute on different types of device, as discussed in Section 2.2. Although it provides portable functionality, its performance will vary drastically across different components of the heterogeneous system. Now, as such systems become more mainstream, they will move from application dedicated devices to platforms that need to support multiple concurrent user applications. Performance variability that may be manageable when the GPU is used as a dedicated acceleration device by a single application poses a problem for concurrent users. Here there is a need to determine when and where to map different applications to best utilise the available hardware resources.

In this chapter, we address the problem of how to schedule multiple OpenCL applications on a CPU+GPU platform. Although scheduling is a much studied subject [Snavely and Tullsen (2000); Zhang et al. (2002); Eyerman and Eeckhout (2010); Augonnet et al. (2011); Singh et al. (2013); Emani et al. (2013)], heterogeneous scheduling is made more complex by the different execution times an application will experience on different devices [Pandit and Govindarajan (2014); Sun et al. (2012); Luk et al. (2009)]. Furthermore, while one application may experience significant performance improvement when moving from a manycore CPU to a GPU, another may experience a slow down. Given a set of application tasks to schedule, it is only possible to determine the best allocation of tasks to devices and their schedule if their execution time is known at schedule time. While this may be possible in certain embedded systems, it is not the case in general purpose systems when the job mix is not known ahead of time. Furthermore, the best schedule can vary depending on optimization criteria; maximizing system throughput may be at the expense of average turnaround time.

We develop a novel scheduling approach which determines at runtime which applications are likely to best utilize a device. We show that speedup is a good heuristic for heterogeneous throughput and develop a novel predictor that determines an ap-

plication's speedup based on static code structure. However, a speedup alone based priority heuristic would favour small jobs with high speedup over longer jobs with more modest speedup. Our scheduler therefore combines speedup prediction and runtime input data size as factors in considering scheduling priority. This technique is applied to a large set of concurrent programs and evaluated for two distinct metrics: system throughput and average normalized turnaround time. We compare our scheduling approach against FluidiCL [Pandit and Govindarajan (2014)], a state-of-the-art kernel split mapping scheme in which both CPU and GPU are fully used. Our approach shows significant performance improvement, for both metrics, over all other approaches.

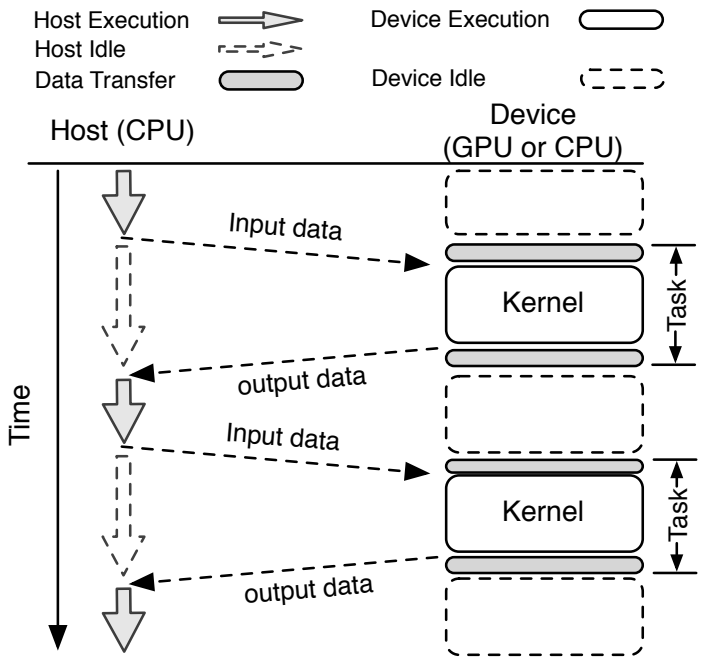
4.2 Background

This work is concerned with the scheduling of multiple OpenCL kernel tasks on a CPU/GPU based heterogeneous platform. A kernel task is referred to as an OpenCL kernel at runtime, which includes computation and associated CPU-GPU communications. This concept is depicted in Figure 4.1(a). Tasks might belong to one or more than one OpenCL program. Note that in this chapter we do not split the work of a single kernel across devices.

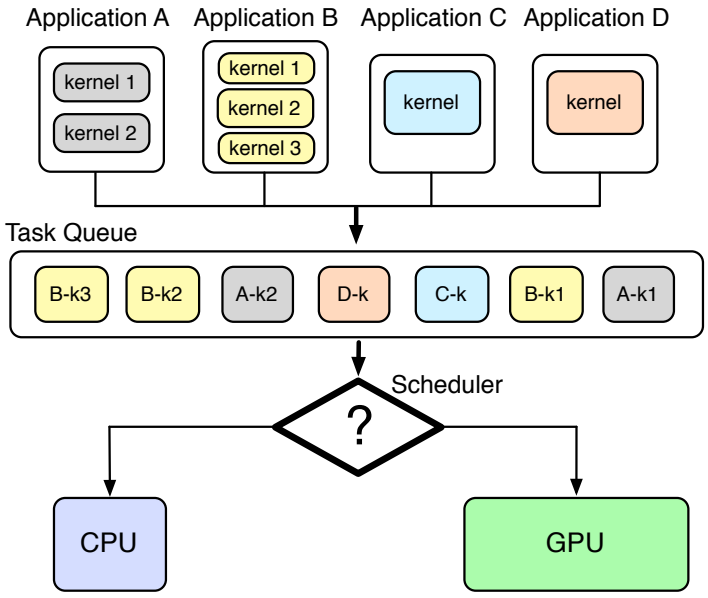
A typical scenario of OpenCL task scheduling is illustrated in Figure 4.1(b). Here we have a task queue that is managed by a runtime scheduler. In this example, the task queue contains several OpenCL tasks submitted by four OpenCL programs, where each task can run on both the CPU and the GPU. It is therefore the runtime scheduler's responsibility to decide which device to use to run a particular task that can lead to the best overall performance (e.g. throughput or turnaround time). This chapter aims to develop a portable approach for efficient OpenCL multi-task scheduling and our goal is to maximize the system throughput without significantly increasing the average application turnaround time. The next section provides an example showing that scheduling program task on CPU/GPU based heterogeneous systems is non-trivial.

4.3 Motivation Example

Consider a scenario of scheduling four OpenCL tasks (kernels) from four OpenCL programs (`bfs`, `BlackScholes`, `Dotproduct`, `QuasirandomG`) on a CPU/GPU heterogeneous system. Figure 4.2 shows the runtime of each individual kernel when it

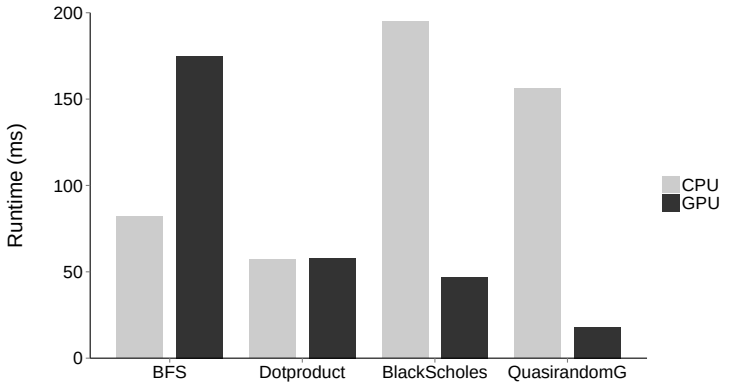


(a) OpenCL tasks

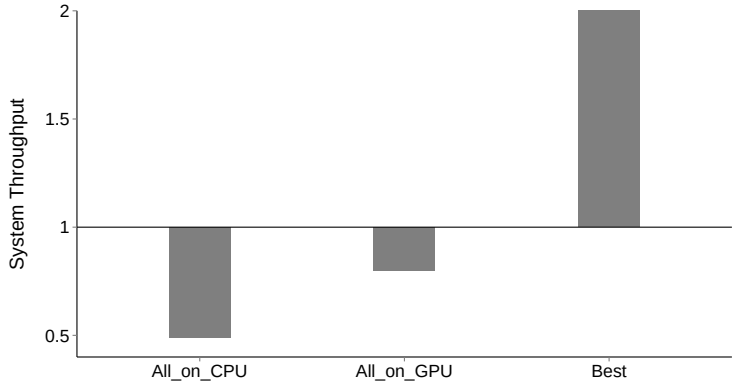


(b) A scheduling scenario

Figure 4.1: Multi-task scheduling on CPU/GPU heterogeneous systems.



(a) Program Runtime



(b) Scheduling performance

Figure 4.2: Task scheduling on heterogeneous systems is challenging – the best scheduling depends on the mix of application tasks and executing on the GPU may not be the best strategy.

runs on the GPU or the CPU. As can be seen from this figure, kernel (`bfs`) is long running on the GPU and the other two (`BlackScholes` and `QuasirandomG`) enjoy significant improvement on the GPU. Conversely, the shortest running kernel on the CPU, `Dotproduct`, shows no improvement when scheduled to the GPU.

We now consider how different scheduling policies allocate these tasks to a CPU/GPU platform and investigate their resulting performance. The first one is a greedy first come first served policy which allocates tasks to whichever device is available `fcfs`. The second policy is to execute the tasks only on the multi-core CPU in a FIFO manner `allcpu`. The third policy runs all tasks on the GPU in a FIFO manner `allgpu`. Finally we consider the best possible schedule if we were to know the program execution time ahead of time `best`. This is impossible in practice but serves as a useful goal for performance.

Figure 4.2 (b) shows the resulting throughput performance. Here we use `fcfs` as our baseline of 1.00 and show the other policies' relative speedup. The `allcpu` scheme is obviously a poor scheme as it only utilizes the CPU. The `allgpu` policy is more effective, but still not able to give performance improvement over `fcfs`. The `best` schedule, however, achieves a speedup of 2x, a significant improvement over the other schemes. Clearly, there is significant room for performance improvement for the policies when compared to the best performance.

This example demonstrates that scheduling policy is critical to system throughput. A good policy depends on whether each individual task can benefit from the GPU execution and how long running the task is. If we know this information before scheduling the tasks, we can then determine efficiently which device to use to run each individual task. What we need is a technique that can predict the GPU speedup of any given OpenCL kernels and estimate the running time of a task. The remainder of the chapter describes how to predict OpenCL kernels speedup and use these predicted speedups together with input sizes as a guide to schedule tasks across the CPU and GPU.

4.4 Overall Scheme

Although knowledge of the execution time of each task is needed for optimal scheduling, accurately determining the execution time of a unseen program is undecidable [Hong and Kim (2010); Zhang and Owens (2011)]. Our approach is to use the predicted speedup of an OpenCL kernel when it is to be executed on the GPU as part of the guide to its scheduling priority. High speedup kernel tasks are scheduled to the

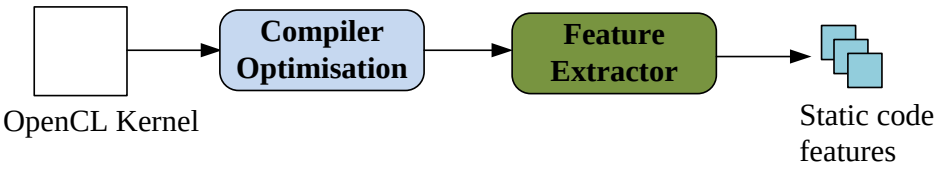


Figure 4.3: Static code features extraction.

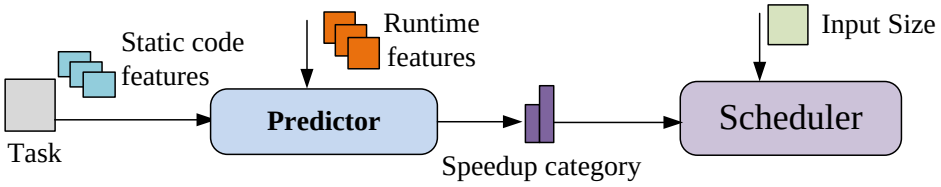


Figure 4.4: Runtime task speedup prediction

GPU, lower speedup ones to the CPU.

Determining the potential speedup of a kernel is non-trivial, so we consider a simpler classification problem. We classify programs into two categories *high*, and *low* speedups and use these classification to assign task priority. Accurately classifying programs in this way relies on the structure of the program and the input data size. Although we have access to the code before execution, the input data size will only be known at runtime. As OpenCL is just-in-time (JIT) compiled, we consider code and input size at the same time.

Figures 4.3 and 4.4 illustrate our 2-part approach. The compiler extracts static code features from the abstract syntax tree for each OpenCL kernel. These features are then combined with runtime data information to predict which speedup category (high or low) this kernel belongs to when running it on the GPU. The prediction is achieved by way of a machine learning model applied to the OpenCL kernel when compiled by the JIT compiler. The prediction, i.e. the speedup category of the input program for a given input, is used by the runtime scheduler to determine which device to use for each individual task.

At the heart of this approach is a speedup category predictor. In the next section, we will describe how a machine learning based classifier can be built to predict the speedup category of any *unseen* OpenCL programs.

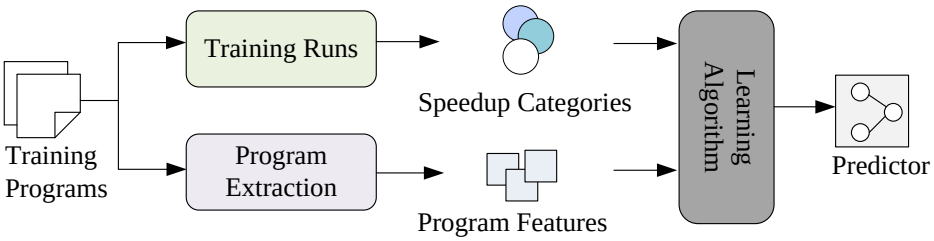


Figure 4.5: The process of training a machine learning predictor with training examples.

4.5 Predictive Modeling

Our predictive model for speedup category prediction is a support vector machine. The input to the model is a set of features that describes the input OpenCL kernel. Its output is a classification that indicates whether the input kernel is a high-speedup or low-speedup kernel.

4.5.1 Building the Predictor

Our predictor is built offline using training programs. The built model can then be used within an OpenCL task scheduler.

Figure 4.5 depicts the process of training a machine learning model using training programs. The training process involves the collection of training data which is used to fit the model to the problem at hand. In our case we use a set of OpenCL programs that are both executed on the CPU and the GPU to measure the speedup of the GPU execution for each individual kernel over the CPU. Depending on the speedup, each kernel will be labelled as either a high-speedup or a low-speedup category. In this work, an OpenCL kernel will be labelled as high-speedup if the measured GPU-speedup is larger than a certain threshold. Otherwise, it will be labelled as low-speedup. This threshold value was determined experimentally, which is set to 4 in this work.

We also extract features for each kernel as described in the following section. The features together with the speedup category for each program from the training data are used to build the model. Since training is only performed once at the factory, it is a *one-off cost*. In our case the overall training process takes less than a day on a single machine.

4.5.1.0.1 Predictive Model Our model is a support vector machines classifier [Bishop (2006)]. We use the Radial basis function kernel, which is able to model both linear and non-linear classification problems. We chose `SVM` as it gives better prediction ac-

Table 4.1: Program Features

Static features	#instructions	#load/stores
	#blocks	#br/condbranches
	#mathFunctions	#vector operations
	#int operations	#float operations
	#control instruction	#logic operations
	#barriers	#atomic operations
Runtime features	local_work_size	global_work_size
	input size	output size

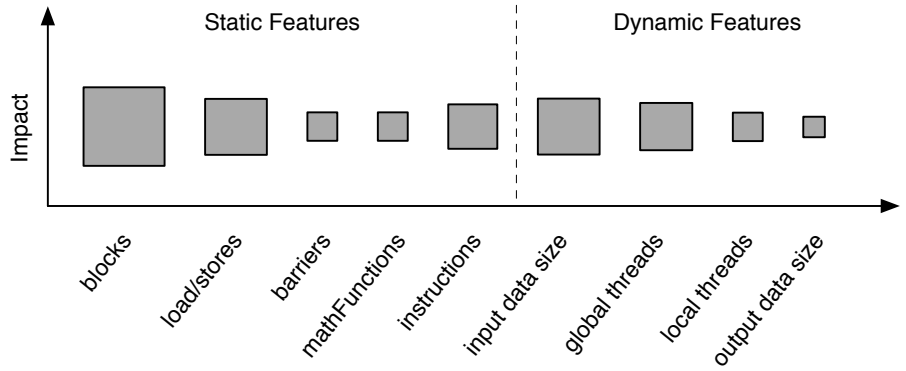


Figure 4.6: Importance of program features. The larger the box, the more important a feature is for the prediction accuracy.

curacy when compared to other models (i.e. K-nearest neighbour and decision trees) in our case.

4.5.2 Program Features

Our predictor uses program features to characterize an OpenCL program. We use both static code features, such as the number of instructions, and parallel runtime parameters, such the number of work items. All the static and runtime features are listed in Table 4.1.

Static code features are extracted from the abstract syntax tree of the OpenCL kernel at the time the program is compiled by the OpenCL just-in-time compiler. The feature extraction tool is based on Clang and LLVM UIUC (2016). At compile time, we extract information about the number and type of operations.

Besides static code features, we also use *parallel runtime features* to characterize the dynamic behavior which is often associated with the program input. The `local_work_size` and `global_work_size` indicate the maximum number of current threads, which are useful for determining the amount of parallelism available. The

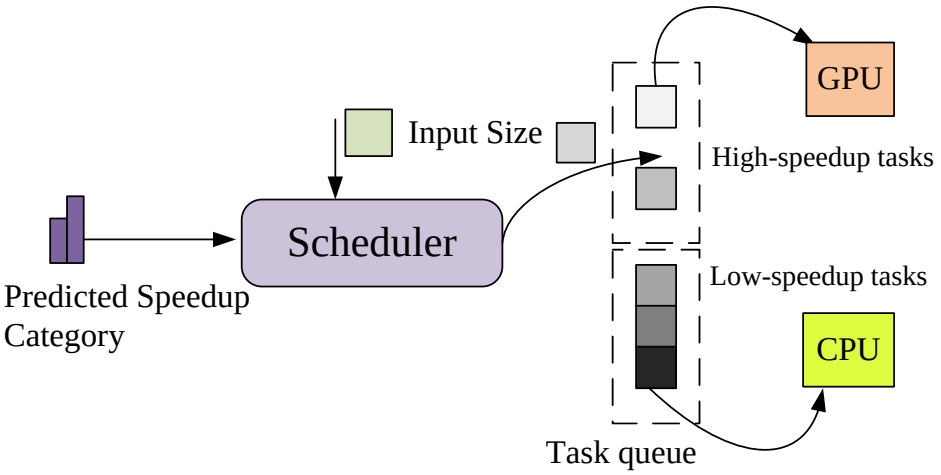


Figure 4.7: New arriving tasks will be inserted into an appropriate position of the task queue based on their predicted speedup categories and input sizes. Tasks from each end of the queue will be dispatched onto the GPU and CPU respectively.

memory transfer represents the communication overhead between the multi-core CPU and the GPU, which can have a significant impact on the GPU speedup. The `input/output size` is estimated by calculating the number of bytes to be transferred between the host CPU and the GPU.

Our predictor is trained with all static features and dynamic features which are shown in Table 4.1. Features which contribute least to the prediction is filtered out by our training process. In our experiment, we only select five static features and four dynamic features. Using fewer carefully selected training features is able to shrink predictor training time without suffering a lost in accuracy. In this chapter, our selected features is shown in Figure 4.6. The overall contribution of each selected feature in our training process is also shown in Figure 4.6.

4.6 Runtime Task Scheduling

Newly arriving OpenCL kernels are inserted into a task queue from which kernel tasks are dequeued and scheduled to either the CPU or GPU when the devices are available as shown in Figure 4.7. The queue is sorted based on the predicted speedup category and program input size. High-speedup kernel tasks are dequeued from one end and scheduled to the GPU, low speedup task are dequeued from the other end and scheduled to the CPU. Tasks will be firstly grouped according to their speedup category where tasks with the same speedup category will be placed together. Those tasks will then be sorted according to the input size in a way that those tasks with relatively

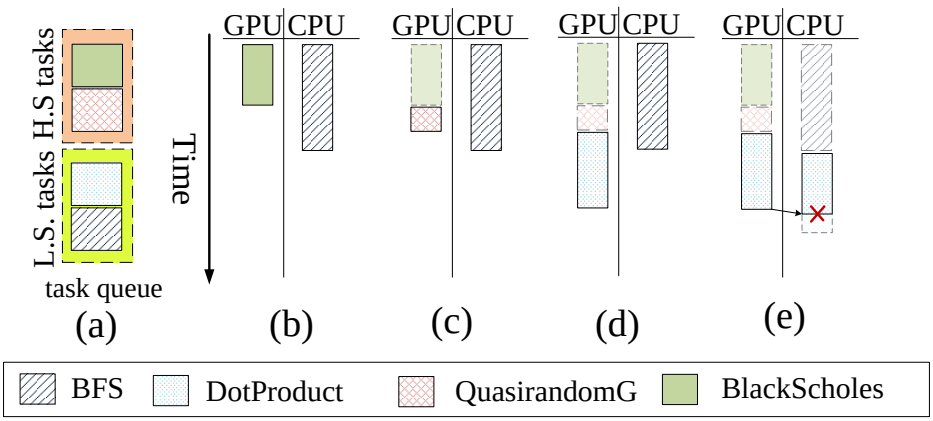


Figure 4.8: The layout of the task queue (a) and the scheduling process (b) (e) for the example shown in Figure 6.1 using our machine learning based scheduler. In this case, our approach achieves the optimal throughput.

smaller input sizes will be placed towards the end of the queue where the CPU will take tasks from. This is because based on our observation tasks with a large input size often correlates with long execution time. We always prefer to schedule tasks that have long runtime but can enjoy GPU execution with a high-speedup onto the GPU.

Since tasks are dequeued from both sides of the task queue, this dequeue process will meet at somewhere in the middle of the queue. The last task of the queue will be replicated and mapped to both the CPU and the GPU. The last task will be first scheduled to an available device and when the other device becomes idle the scheduler will map a duplicated copy to this device. The scheduler then waits for one of the tasks to complete and kills the outstanding duplicate.

Scheduling Example: Figure 4.8 shows how the four tasks, `bfs, Dotproduct, BlackScholes, QuasirandomG`, presented in Figure 6.1 are scheduled by our scheduler. Our predictor takes the feature values for each OpenCL kernel task and predicts to use the CPU or GPU for scheduling. For instance `BlackScholes` is classified by our predictor as a high-speedup category, it is scheduled on the GPU. `bfs` has a set of different feature values, which is classified by the predictor as low-speedup task, it is scheduled on the CPU. Both `BlackScholes` and `QuasirandomG` are classified as high-speedup tasks and the other two are classified as low-speedup tasks. Based on their speedup categories and input sizes, the tasks are sorted in the task queue as shown in Figure 4.8 (a). If we assume both the GPU and CPU are available upon the time those tasks arrive, this will result in a scheduling plan as depicted in Figures 4.8 (b) - (e) over time. For this example, our scheduler gives the best throughput performance.

4.7 Alternative Policies

4.7.1 Alternative Scheduling Policies

We compare our approach against four different strategies:

- **All_on_CPU**. Using this scheme, tasks are dispatched to the shared CPU in the arriving order.
- **All_on_GPU**. Using this scheme, tasks are dispatched to the shared GPU in the arriving order.
- **FCFS**. This is a first come first served approach. Using this scheme, application tasks will be put into the task queue in the order as they arrive. Then tasks will be dispatched to any available computing device (either the GPU or the CPU).
- **Input size guided**. In the task queue, tasks are sorted based on the amount of bytes needed to be transferred from the CPU to the GPU. With this scheme, the GPU always gets a task that has the largest input and the CPU always gets a task with the smallest input.
- **Work item guided**. In the task queue, tasks are sorted according to the number of global work items of the kernel. Using this scheme, the GPU always gets a task that has the largest number of work items while the CPU always gets a task with the smallest number of work items.

There are some alternative scheduling schemes, such as the shortest-job-first scheme [Pinedo (2008)], which all require to know the task execution time ahead of time. Since our experimental settings assume this information is not available to the scheduler, those approaches cannot provide a fair comparison and hence are not included. Round Robin is another widely used task scheduling scheme. However, because the current GPU implementation does not support context switch or preemption, this is not available for comparison either.

4.7.2 Partitioning OpenCL Kernels across Devices

The FluidiCL [Pandit and Govindarajan (2014)] runtime utilizes both the multi-core CPU and the many-core GPU to concurrently execute a single OpenCL kernel. In this way, the CPU executes part of the kernel, starting from the upper end of the working

Table 4.2: Hardware platform

	Intel CPU	NVIDIA GPU	AMD GPU
Model	Core i7 2600K	GeForce GTX 590	Radeon HD7970
Core Clock	3.4 GHz	1215 MHz	1000 MHz
Core Count	4 (8 w/ HT)	1024	2048
Memory	8 GB	3 GB	3GB
Memory Bandwidth	21GB	327 GB	288 GB

space, while the GPU executes the whole kernel, but starting from the lower end of the working space. When the GPU reaches a work-group that has already been executed by the CPU, the whole kernel execution is considered to have been completed and the results will be merged. However, this scheme can only apply to one single OpenCL kernel. As a result, kernels from multiple applications will have to be executed sequentially. Furthermore, distributing work items between the CPU and GPU requires synchronization and communications between the two devices, which can incur significant runtime overhead. We compare our approach against FluidiCL in Section 4.9.2

4.8 Experiment Setup

This section describes our experimental setup and the evaluation methodology used in the remainder of the chapter.

4.8.1 Platform and Benchmarks

Platform and Software Tools: We evaluate our approach on two CPU-GPU systems: both use an Intel Core i7 4-core CPU. One system contains an NVIDIA GeForce GTX 590 GPU, the second an AMD HD 7970 GPU. Both run with the OpenSUSE 12.3 Linux. Our compiler is GCC 4.7.2 with -O3 as the compiler option. We use the NVIDIA CUDA Toolkit 3.1 which has an OpenCL just in time compiler. Details of the hardware platforms are shown in Table 6.1.

Benchmarks: We used 35 different benchmarks from three mainstream OpenCL benchmark suites: the NVIDIA OpenCL SDK v4.2, the AMD SDK v2.8 and the Parboil OpenCL benchmark suite v2.5. From the above benchmark suites, we selected those applications whose input data size can be easily scaled. Also, to compare our method against FluidiCL, we included the Polybench benchmarks that can experience enhanced performance by FluidiCL partitioning in its work. In the experiments, we ran

Table 4.3: Benchmarks and Input Sizes

Suite	Benchmarks	Input Size	Benchmark	Input Size
NVIDIA	BlackScholes	12K - 12M	ConvolutionSeparable	1.6M - 420M
	DXTCompression	17M - 604M	DotProduct	41M - 654M
	FDTD3d	452M	HiddenMarkovModel	69M
	Histogram	67M - 268M	MatrixMul	63M
Parboil	BFS	64M	Cutcp	3M - 36M
	Sgemm	192K - 12M	Spmv	49K - 31M
AMD	BinarySearch	2K	BinomialOption	3K
	BitonicSort	16K - 65K	BlackScholes	1M - 4M
	BlackScholesDP	2M - 5M	DCT	16K - 16M
	DwtHaar1D	4K - 65K	FastWalshTransform	4K - 131K
	FloydWarshall	262K	Histogram	4M-1G
	MatrixMultiplication	16K - 1M	MatrixTranspose	16K - 67M
	PrefixSum	2K - 16K	QuasiRandomSequence	1K
	Reduction	8K	ScanLargeArrays	4K - 65K
	SimpleConvolution	16K		
Ploybench	ATAX	1G	BICG	1G
	CORR	50M	GESUMMV	1G
	SYR2K	50M	SYRK	33M

each benchmark with a range of different inputs. The list of benchmarks and inputs is shown in Table 4.3.

4.8.2 Runtime Scenarios

Our evaluation setting consists of multiple runtime scenarios with 49 different task mixes where each task mix contains 2 to 50 OpenCL kernels (tasks). The task mixes are grouped into three task groups with different numbers of tasks: *small*, *medium*, and *large*. We consider a task group to be small, medium and large if it contains less than 10, 10-20, or more than 20 (up to 50) kernels respectively. For each task mix, we tried up to 125 different task combinations with different OpenCL kernels and input sizes. We report the average performance per task group across all combinations. The OpenCL applications of each task group were randomly selected from the list of benchmarks given in Table 4.3. Moreover, in the experiments we replayed each scheduling decision 10 times and calculated the average performance of each decision to reduce the impact of jitter. Finally, we assumed all tasks arrive at the same time and have the same priority.

4.8.3 Performance Evaluation

Performance Metrics: To evaluate our approach, we used two metrics, *system throughput*, a system oriented metric, and *turnaround time*, a user oriented metric. Those two metrics have widely been used to evaluate the performance of a scheduler in a multi-tasking environment [Snavely and Tullsen (2000); Eyerman and Eeckhout (2010)]. Our goal is to maximize the system throughput which in general leads to favourable turnaround time results. The definitions of the two metrics are given as below.

System throughput (STP) is a *higher is better* metric. It describes the number of tasks completed per unit time. This is calculated by using the fcfs scheme as a baseline of 1.0, showing the relative speedup of other scheduling policies. It is defined as

$$STP = \frac{\sum T^i_{FCFS}}{\max(\sum T^m_{cpu}, \sum T^n_{gpu})} \tag{4.1}$$

where T^i_{FCFS} is the execution time given by `FCFS`, and T^m_{cpu} and T^n_{gpu} are the execution time by running task T^i on the CPU and the GPU respectively.

Average normalized turnaround time (ANTT) is a *smaller is better* performance metric. It quantifies the time between a task is created and its completion, indicating the average user-perceived delay in multi-tasking environment compared to running a single task on the system. In the experiments, the turnaround time is normalized to the fcfs scheme. ANTT is defined as:

$$ANTT = \frac{1}{n}\sum_{i=1}^{n}\frac{T^i_{sch}}{T^i_{FCFS}} \tag{4.2}$$

where T^i_{FCFS} and T^i_{sch} are the time between task T^i is created and its completion using fcfs and an alternative scheduling policy respectively.

Predictive Modeling Evaluation We use *leave-one-out cross-validation* to train and evaluate our predictive modeling based scheduler. This means we remove the target OpenCL programs to be predicted from the training program set, collecting training examples without the target programs to be presented, and then learning a model from the training examples. It is a standard evaluation methodology, providing an estimate of the generalization ability of a machine learning model in predicting unseen programs.

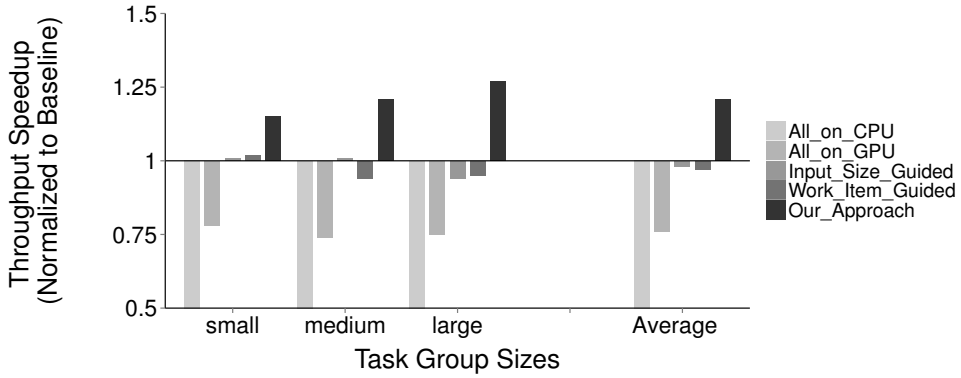
4.9 Results

In this section we present the experimental results of our approach. The baseline is `FCFS`. Our goal is to maximize the STP and minimize the ANTT, so that the system can finish as many tasks as possible per time unit and at the same time reducing the turnaround time.

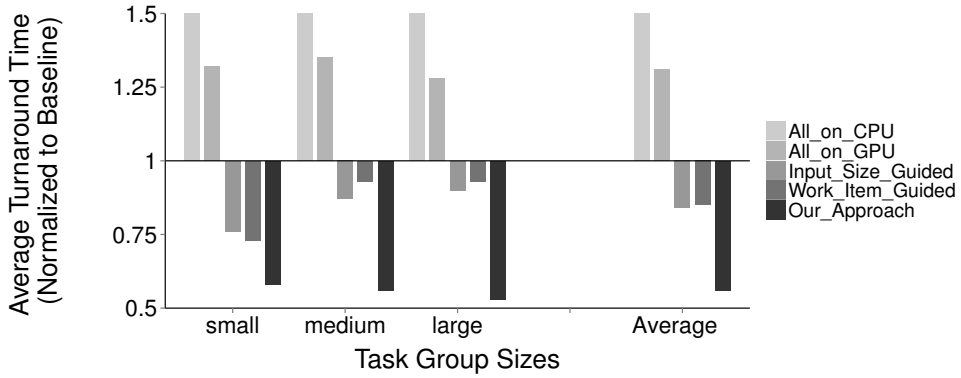
4.9.1 Overall Results

STP: As can be seen from Figures 4.9 (a) and 4.10 (a), our approach consistently outperforms the baseline for this higher is better metric. As the number of tasks to be scheduled increases, we see an overall increase in the STP. The baseline `FCFS` scheme performs well for a small task group and all other alternative approaches give slowdown performance except for our approach where a 1.1x improvement is observed on both platforms. The improvement of our approach increases to an average STP of 1.4 for a large task group. This is not an unexpected result as `FCFS` simply allocates a task to the first available device without considering which the most appropriate computing device is. This scheme may work well for a small task group as the number of available scheduling options is small, but is unlikely to achieve good performance as the number of tasks to be scheduled increases where a large number of scheduling options is opened up. By assigning high-speedup tasks to the GPU and low-speedup ones to the CPU, our approach can make effective use of both GPU and CPU and achieves higher throughput. On average, our approach achieves a throughput of 1.25 across all task group sizes. This significantly outperforms other approaches which all fail to improve the STP.

ANTT: Figures 4.9 (b) and 4.10 (b) show the achieved ANTT, a lower is better metric, on the NVIDIA and AMD platforms respectively. As can be seen from the diagrams, our approach not only improves throughput and but also the ANTT. Our approach consistently outperforms the baseline and has a lower ANTT as the number of tasks to be scheduled increases. The ANTT given by our approach is 0.58 and 0.8 for a small task group on the NVIDIA and the AMD platforms respectively, which is further reduced to less than 0.6 for a large task group. Similar to the STP results, our ANTT performance is improved as the number of tasks to be scheduled increases where the number of available scheduling options increases. Overall, our approach performs well with an average ANTT of 0.56 and 0.65 on the NVIDIA and the AMD platforms respectively. On average, our approach outperforms all other approaches by



(a) System throughput on the Nvidia Platform

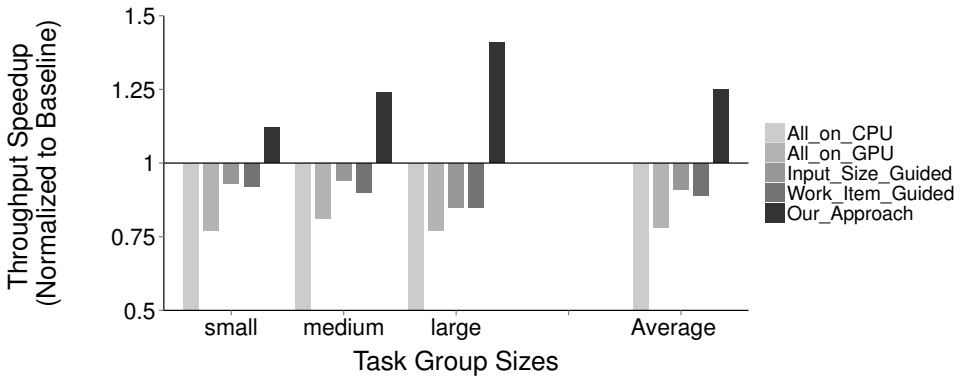


(b) Normalize average turnaround time on the Nvidia Platform

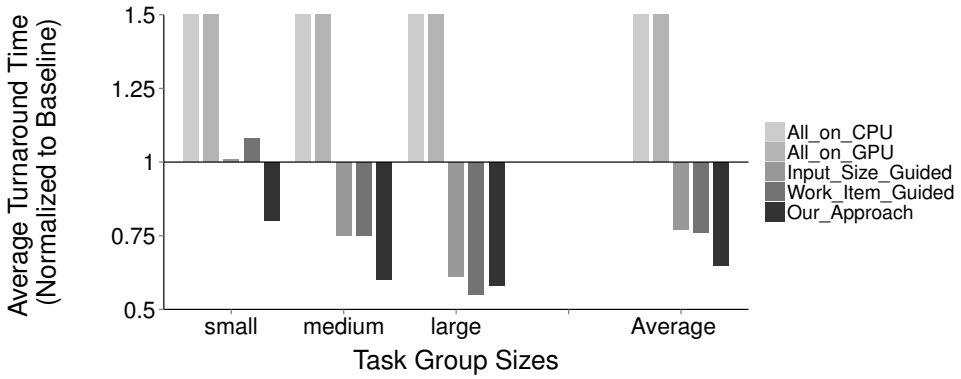
Figure 4.9: The achieved STP (a) and ANTT (b) on the Nvidia GPU platform. Our approach achieves, on average, a 21% and 56% of improvement over the baseline (`FCFS`) for the STP and ANTT metrics respectively.

reducing the turnaround time by at least 1.5x and 1.2x on the NVIDIA and the AMD platforms respectively.

Summary: Our approach constantly outperforms all alternative approaches for two performance metrics: STP and ANTT. The advantage of our approach is largely attribute to its capability to predict the potential speedup category of each kernel task. Without this information, the alternative schemes may inappropriately assign tasks onto the GPU, which may not be able to benefit from the GPU execution. This leads to the poor GPU utilization and overall poor scheduling performance.



(a) System throughput on the AMD Platform



(b) Normalize average turnaround time on the AMD Platform

Figure 4.10: The achieved STP (a) and ANNT (b) on the AMD GPU platform. Our approach achieves, on average, a 25% and 65% improvement over baseline (`FCFS`) for the STP and ANTT metrics respectively.

4.9.2 Comparison to State-of-the-Art

Figure 4.11 (a) compares the STP improvement achieved by our approach against FluidiCL on the NVIDIA platform. The performance of FluidiCL is disappointing when scheduling multiple OpenCL tasks. It gives an average slowdown of 0.93 over `FCFS`. Only for small task groups, by partitioning the work of OpenCL kernels across the CPU and the GPU, FluidiCL is able to achieve a modest speedup (1.03x) over `FCFS`.

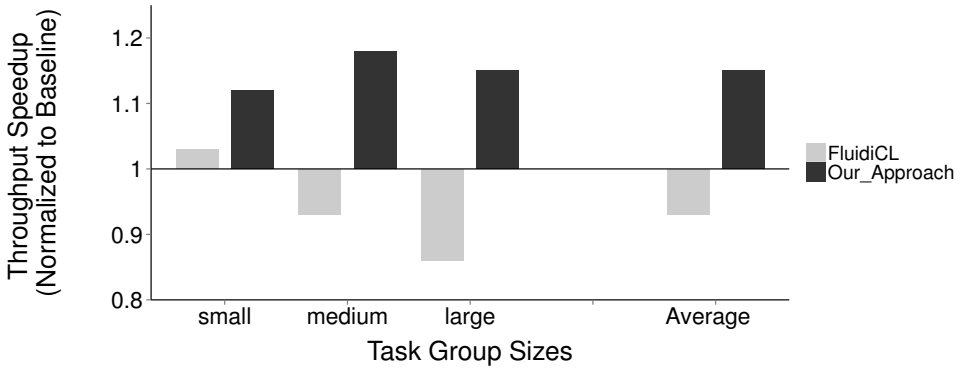
Using FluidiCL, a faster computing device will eventually execute a large portion of the work. For all the OpenCL kernels we used in the experiments, there is always one computing device (either the CPU or the GPU) which clearly outperforms the other. As a result, the use of an additional computing device rarely accelerates the computation of a single kernel. Also, we observed that FluidiCL often introduces expensive synchronization and communication overhead between the two computing devices for distributing work and merging results, leading to overall slowdown performance. Unlike FluidiCL, our approach avoids such synchronization and communication overhead, giving constantly better STP performance over FluidiCL. On average, our scheme improves the STP by 1.23x compared to FluidiCL. Furthermore, our approach is able to minimize the average turnaround time by 1.96x over FluidiCL (0.55 vs 1.08 in Figure 4.11 (b)).

4.10 Analysis

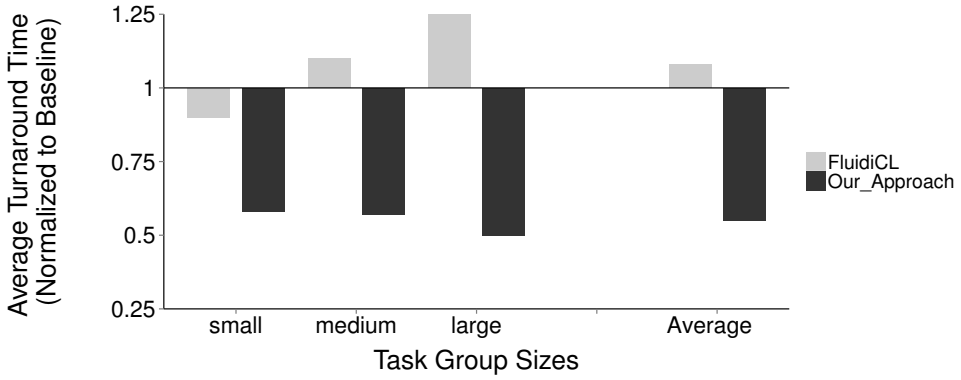
4.10.1 Best Available Performance

Although our scheme performs well compared to alternative approaches, it is useful to know whether there is any further room for improvement. It may be the case that the competitive schemes are very poor and that a smarter scheme could perform significantly better. In Figure 4.12, we compare our scheme against the best available STP performance on the NVIDIA platform. This is obtained by exhaustively trying all possible scheduling options. This `best STP` scheduling is unrealistic in practice, but provides a useful upper bound.

When there is a small number of kernel tasks to schedule, our approach gives nearly optimal performance. When the number of kernel tasks is large, there is room for improvement. The `best STP` schedule is able to improve performance by 50% for the STP. This is because speedup is only a proxy for execution time. Errors in estimating



(a) System throughput of our approach vs FluidiCL



(b) Normalize average turnaround time of our approach vs FluidiCL

Figure 4.11: Our approach constantly outperforms FluidiCL with an average improvement of STP (a) (1.15x vs 0.93) and ANTT (b) (0.55 vs 1.08) on the NVIDIA platform.

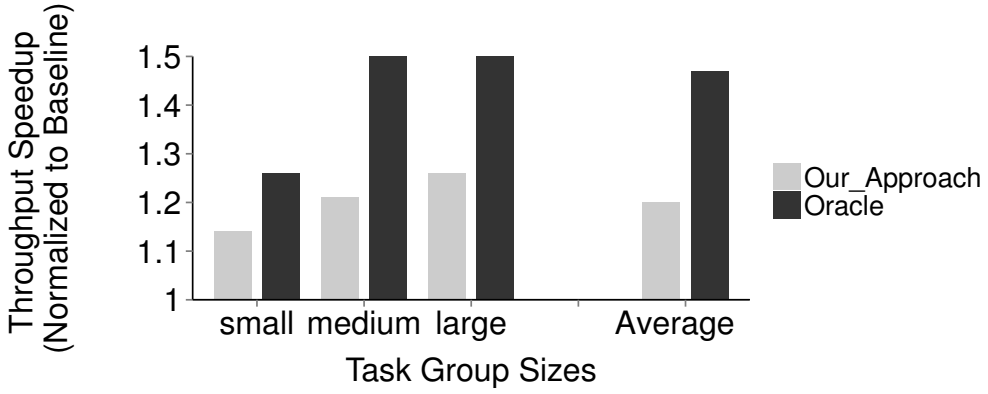


Figure 4.12: Comparison to the best available STP. Our approach achieves 43% of the best available performance on the NVIDIA platform.

execution time will increase as the number of tasks increase. In fact while we can determine the `best STP` schedule, it is impossible to determine accurately the best ANTT schedule due to combinatorial complexity. While we will never achieve the performance of the `best` schedule as it is undecidable, there is still room for improvement. In the next experiment, we therefore evaluated prediction accuracy and see if this has an impact on performance.

4.10.2 Impact of Prediction Accuracy

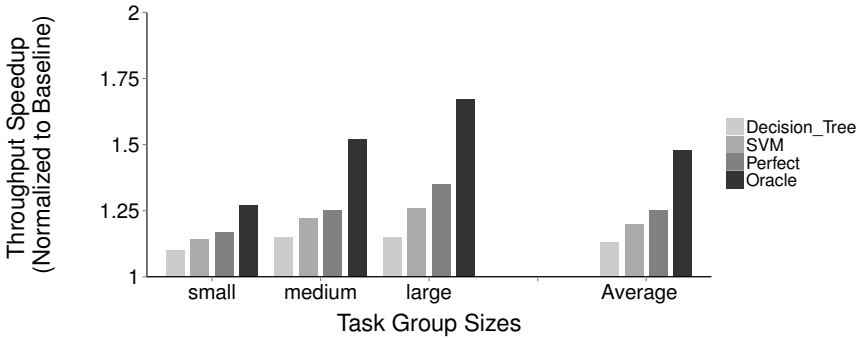
We would like to know how the prediction accuracy affects the scheduling performance. To do so, we have also considered a decision tree based model and a theoretically `Perfect` predictor which always gives the correct speedup classification (i.e. the prediction accuracy is 100%).

Figure 4.13 compares the STP and ANTT performance achieved by the three models. As can be seen from the diagram, prediction accuracy has significant impact on the scheduling performance and in fact the more accurate a predictor is the better performance the scheduler has. The SVM model has higher accuracy (87%) than the decision tree model (72%) for STP (Figure 4.13 (a)). The SVM model therefore gives constantly better results for both evaluation metrics when compared to the decision tree model (1.2x vs 1.13x). A `Perfect` classifier further increases performance to 1.25x. We see a similar pattern for ANTT (Figure 4.13 (b)). Here the SVM model gives an ANTT of 0.57. The decision tree once again degrades performance to 0.62 while the `Perfect` predictor improves it. Although building a `Perfect` predictor is almost impossible in reality, this experiment result confirms that the scheduling performance can be further improved with a more accurate model. However, by comparing to the best available performance of STP[1] there is still room for performance improvement even for the *perfect* predictor. One reason may be that our current approach only has two speedup categories which essentially is a coarse-grained classification. This hypothesis is confirmed by the experiment described in the next section.

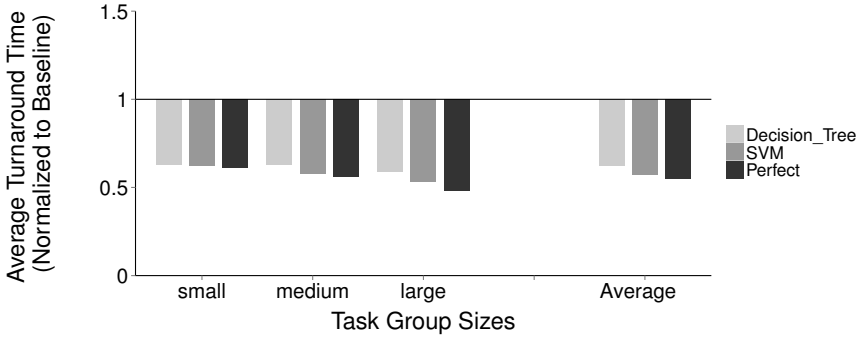
4.10.3 Fine-grained Speedup Categorization

Currently, each kernel task is classified into two speedup categories. We want to know whether a finer-grained classification could improve scheduling performance. To do

[1]Given the extremely large combinatorial scheduling option available, it is infeasible to find the best ANTT performance. Therefore, we do not present the best ANTT performance.



(a) System Throughput



(b) Normalized Average Turnaround Time

Figure 4.13: The STP (a) and ANTT (b) achieved by different predictive models. The more accurate the model is the better the scheduling performance is.

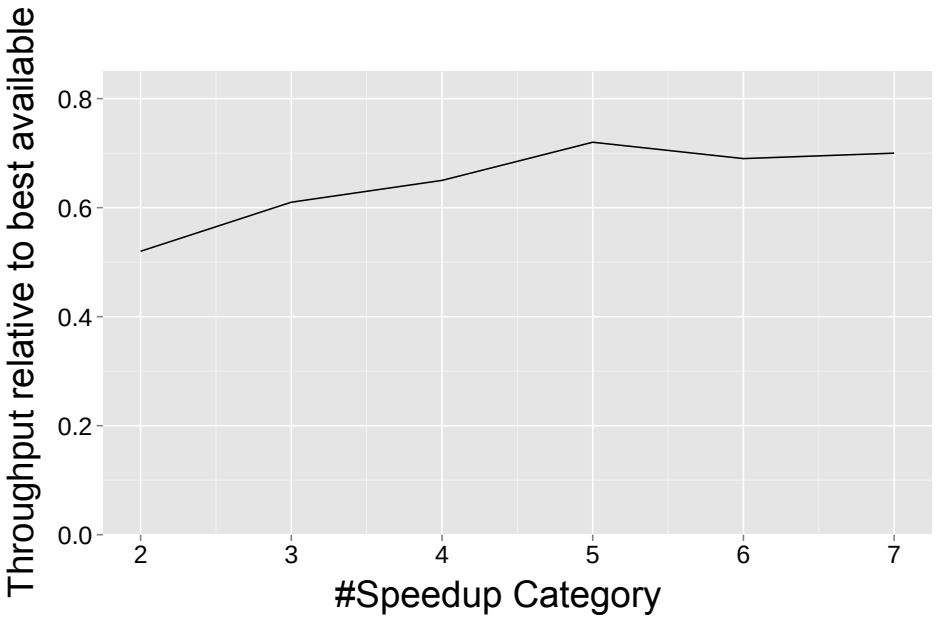


Figure 4.14: STP performance tends to improve with a finer-grained speedup category classification.

so, we first break down the number of speedup categories to create 3-7 categories; we then classify the speedup of each kernel task using the actually measured speedup. Figure 4.14 shows the STP performance using different category granularities. As can be seen from this figure, a finer-grained classification in general leads to better STP performance. This shows that our approach can be further improved using finer-grained classification and this is our future work.

4.10.4 Overhead

Our predictive model is trained offline with training examples. In this work, collecting the training examples took less than a day using a single machine, which has no impact on runtime cost. The overhead of using the trained model includes extracting program features and making predictions. This overhead is negligible (approximate 10ms in total), which has been included in all experimental results.

4.11 Conclusions

This chapter has presented an efficient OpenCL task scheduling scheme which schedules multiple programs across CPUs/GPUs heterogeneous platform. Our scheduler uses a speedup predictor and runtime input data size to schedule tasks. This technique

is applied to a large set of concurrent programs where it shows significant performance improvement over all other existing approaches. This work shows that speedup is a good priority function. Future work will investigate improving speedup prediction accuracy using larger training data sets. Significant further improvement is potentially available when using more fine-grained speedup classification.

Chapter 5

Concurrent Kernels

This chapter presents a method for efficient concurrent kernel execution on CPU-GPU platforms. By space sharing the GPU, we improve utilization and overall system performance. We propose a kernel merging method that gives fine grain and accurate control of space sharing on GPUs by two OpenCL kernels. As the co-execution of kernels does not always provide good performance, we propose a machine learning-based technique to select pairs of kernels that can benefit from co-execution based on their static code characteristics and dynamic parameters.

This chapter is structured as follows: Section 5.1 introduces the background of concurrent kernel execution. The architectural difference, which is discussed in Section 2.1, between CPU and GPU, requires a distinctive technique to increase GPU utilisation hence improve system performance. Some of the concurrent kernel methods that are reviewed in Section 3.2.1 have been discussed in this section again, as being the state-of-the-art approach, they provide the best performance. We will compare our approach with them in the following sections. Section 5.2 provides motivation showing that the co-execution of carefully selected kernels improves performance; otherwise, performance degrades. Section 5.3 describes how concurrent kernels are constructed by static merging. Section 5.4 and 5.5 presents our experimental setup and the experimental results. Section 5.6 provides a detailed analysis of our approach. Finally, Section 5.7 summarizes the chapter.

5.1 Introduction

GPUs and CPUs are architecturally different. CPUs are latency-oriented systems with few cores but large amounts of cache memory. They use sophisticated branch predic-

tion and speculative fetching logic to increase instruction-level parallelism. GPUs by contrast, are throughput-oriented systems. Instead of focusing on per thread performance improvement, they increase throughput by massive parallelism. Each generation of GPU has more computing units, larger register files and shared memory than its predecessor to execute a greater number of threads. More hardware resources usually mean higher performance, however, it can lead to a low utilization. Low resource utilization restricts the expected performance brought by hardware development, as only a fraction of the hardware is utilized at any given time. Kernels running on the GPU are constrained by various limitations. Normally, they exhaust one of the resources, e.g. registers, but leave the others underutilized, e.g. local memory.

The core scheduling unit, namely the workgroup, contains a group of threads that execute concurrently and share local data. Several hardware constraints limit how many threads each block of a launched kernel can have. These hardware limitations include the maximum thread numbers, maximum dimensions of each workgroup, registers, and local memory consumptions. Usually, only one of the above requirements determines the size of the workgroup, leaving the others underutilized. Hardware underutilization restricts the system from approaching its peak performance.

Concurrent kernel execution is a promising solution to increasing GPU utilization. As different kernels have various resource requirement, running multiple instances on the same GPU at the same time can optimize the hardware configuration since some of the kernels could use the spare resources left by others. One of the best-known works is Elastic Kernel (EK) [Pai et al. (2013)], which scales the workgroup size to make multiple kernels fit the same GPU. By launching through different CUDA streams, the blocks, or workgroups, from candidate kernels arrive concurrently, then run in parallel. Since the hardware executes various kernel functions, its utilization is improved.

Concurrent execution enhances resource utilization then boosts system performance only if the co-running kernels are carefully selected. If multiple kernels are competing for the same hardware resource and leave others underutilized, overall utilization is poor, and performance will be even worse than running those kernels sequentially. Therefore, a smart selection is necessary when we adopt the approach of concurrent kernels execution.

To select the kernels to run in parallel, Energy-Efficient Concurrent Kernel (EECK) [Jiao et al. (2015)] profiles each candidate application ahead of time to acquire runtime features. After training from these features, a neural network model determines which

kernels run together and which do not improve energy and throughput at the runtime. Since those features cover a broad range of attributes of the application, the predictive model has useful information about the kernels to find good combinations. While this approach works well, it introduces a substantial runtime overhead in acquiring those crucial features.

Prior work limitation There are two significant problems in prior work. The first is that they are all profiling-driven approaches, requiring running the applications in advance. The cost brought by this method is not trivial and is not feasible in a dynamic environment with unknown user jobs. Therefore, profiling ahead of time then executing is unrealistic in practice for multi-user applications. Secondly, prior work issues workgroups of distinct kernels to the GPU alternately to perform concurrent execution, which is an approach that is not accurate enough in practice. The assumption of this method is that the hardware block scheduler dispatches workgroups to the computing units by a round-robin algorithm, which is unfortunately is not always true. Indeed, the launching order of workgroups, is undefined [NVIDIA (2015)]. Also, the approach of interleaving kernel workgroups has a problem of adaptivity, as workgroups of separate kernels have a various length of execution time on different GPU devices. Therefore, workloads could easily execute sequentially if the time for workgroup execution is shorter than the API function call for issuing them.

New Approach To solve the above problems, we propose a smart concurrent kernel execution in this chapter. Instead of profiling every candidate application ahead of time to find the best co-running combinations, we use an offline trained model to classify those targets according to their predicted associativities. Then, rather than issuing kernels via independent command queues, or streams, at runtime, we merge the target kernels by a source-to-source JIT compiler and launch the newly created kernel on the GPU as a replacement. In our approach, we use static features and runtime parameters to train the offline model and classify the new arriving jobs. Hence, there is no need for new jobs profiling. This approach outperforms existing techniques and in the following parts of this chapter, we present the details of the implementation and the result analysis.

5.2 Motivation

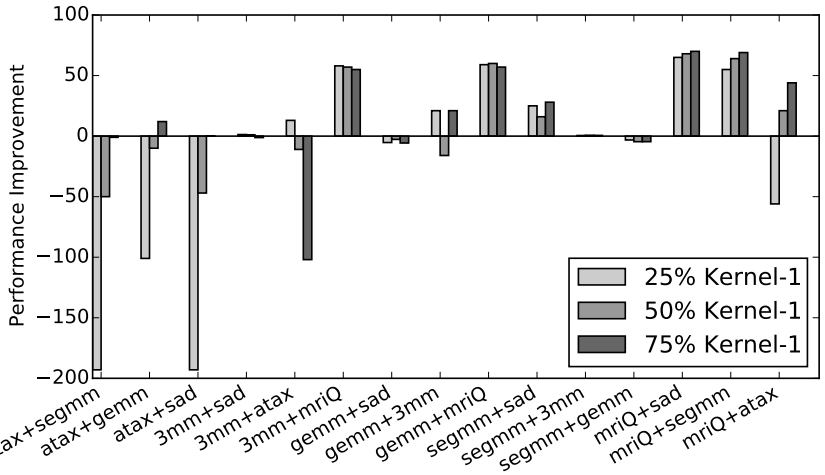
Concurrent execution of kernels on a GPU can enhance hardware utilization thereby improving performance. However, the improvement depends on the programs selected and how much of the GPUs resources are allocated to each. The amount of GPU resource allocated to each kernel is called the mixing proportion. To demonstrate the impact of kernel selection and mixing proportion, we run all pairwise combinations from six benchmarks, three of them from Polybench and the other three from Parboil benchmark suite on AMD 7970 GPU, with a Tahiti architecture.

Figure 5.1 shows the results. Here the x-axis denotes the pair of programs run together, and the y-axis describes the performance improvements over running the kernels non-concurrently. Each pair has three performance bars. The first corresponds to 25% of the GPU allocated to the first program with the rest assigned to the second program. The second and the third bar represent 50% and 75% respectively. The performance of many concurrent kernels is poor and only improves in certain cases.

In figure 5.1, no matter how `atax` and `sgemm` are combined, their performance is always worse than running these two kernels sequentially. Kernel pairs, such as `3mm+mriQ`, `sgemm+mriQ`, `sgemm+sad`, and `mriQ+sgemm`, experience a higher throughput when running them concurrently. Other kernel pairs, such as `3mm+atax`, `gemm+3mm`, and `mriQ+atax`, can achieve good performance, but the correct allocation of resources is critical. Otherwise, they will slow down.

Sharing resource among kernels is complex. The widely accepted idea is that the intensity of computation and memory access have a significant impact on resource sharing. Therefore, the mix of compute-bound and memory-bound applications would be preferable for the sake of performance.

To examine how performance is affected by the concurrent execution, we profile all those candidate kernels and list the profiling information in Table 5.1. There are two columns showing the computing and memory accessing intensity respectively. Apart from the time spending on computation and memory access, the rest of the time is cost by the hardware unit stalls and conflicts, such as write unit stall, memory unit stall, fetch unit stall, and memory bank conflict. As these kinds of information are very low level and vary with the particular hardware resource configuration as well as the characteristics of the input data, they are excluded from our experiment. The data in each column reports the percentage of GPU time the computing unit and memory unit is active. Take `atax_kernel1` for example; the computing unit is active for 0.315%



(a) Performance improvement by concurrent kernel execution. (result in percentage)

Figure 5.1: Multi-Kernel execution on CPU+GPU platform

of the GPU time, and the memory unit is busy for 88.86%. Therefore, we denote `atax_kernel1` as a memory bound application. We list all the six applications according to their computation intensity ascendingly. Most of the applications are memory bounded except `sgemm` and `mri-q`.

Table 5.1: Compute vs Memory Intensity

Kernels	Compute Intensity	Memory Intensity	Benchmark Suite
atax_kernel1	0.315	88.86	Polybench
sad_calc_8	1.7	90.27	Parboil
sgemm	7.58	32.43	Parboil
3mm_kernel2	20.83	87.41	Polybench
gemm_kernel	23.3	88.9	Polybench
mri-q	35.16	0.16	Parboil

Combining computing-intensive and memory-intensive application can improve in some cases, but not always necessarily so. As a computation-bounded application, `mri-q` is ALU heavy but rarely accesses memory. Co-running `mri-q` with other

memory-bounded application, like sad, could improve performance, no matter how the two kernels are mixed. However, if we co-execute mri-q with another memory bound kernel, such as atax, the mixing ratio is critical, though according to profiling atax is as memory-bound as sad. From figure 5.1, we can see, the greater proportion mri-q has, the better overall performance is. For an application like sgemm, co-running with mri-q improves performance perhaps because it has low pressure in computation and memory accessing. But, on the other hand, applications like 3mm and gemm, which are balanced workloads and both have heavy usage of ALU and memory, can still benefit from co-execution with mri-q, no matter how they mix with mri-q. Finally, as a low resource pressure workload, sgemm can experience better performance from co-execution with some other applications, such as mri-q and sad, but it can also suffer a slowdown with workloads, like atax and gemm.

All in all, concurrent kernel execution is a promising approach to improve system performance by strengthening hardware utilization. However, selecting the best co-execution candidates is a non-trivial task, as the widely accepted idea of mixing computing-intensive with memory-intensive kernels does not always work well, and in some cases, it degrades.

5.3 Concurrent Kernel Construction

There are two main approaches, workgroup slicing and kernel fusion, that enable multiple kernels to run concurrently on GPU. Though modern GPUs support concurrent kernel execution, the default strategy is back-to-back execution. The first few workgroups of the next kernel can share some spare computing units if the last few workgroups of the current kernel cannot exhaust all of them. Concurrency only occurs at the moment of one kernel replacing another, and most of the time all hardware is occupied by a single kernel. Workgroup slicing and kernel fusion are two widely used approaches to bypass the limitation of the back-to-back concurrency.

Workgroup Slicing

Workgroup slicing, or block slicing, is a general technique to share a GPU among distinct kernels. The overall workgroups have been separated into several slices, where each slice holds an equal number of workgroups. The number of workgroups within each slice is no more than the number of SIMD processors, so as to leave some hard-

ware to fit other kernels. Launching workgroup slices through separate command queues makes the candidate kernels run in parallel. By using different sized slices, kernels share the hardware in proportion.

Workgroup slicing works under the assumption that the hardware schedules workgroups in a First-In-First-Out (FIFO) strategy. Unfortunately, the precise order of workgroup scheduling is not defined. If the execution time of a workgroup is faster than the dispatching API call, only a proportion of computing units is active at any given time. This leads to worse utilization compared to the default back-to-back concurrent strategy.

Kernel Fusion

The alternative approach is kernel fusion, a static method that fuses candidate kernels to create a single new one, then dispatches it to the GPU. Kernel fusion is a popular method, particularly for kernels with data dependencies [Lutz et al. (2015)]. If one kernel's input is another kernel's output, fusing these two kernels could eliminate extra data movement between the main memory and GPU memory. Also, data locality could be improved, as the code for processing another kernel's output has a higher chance of finding the needed data in its cache.

The system environment has less impact on fused kernels when compared to its slicing counterpart, as the kernel is issued to the device only once. Launching the kernel once, instead of iteratively issuing slices one after another, exposes fewer chances for the operating system to preempt its workgroup dispatching. Also, the sequential executing of workgroups caused by short workgroup execution time does not occur. As the new kernel is created by source-to-source transformation, this process is not visible to the hardware.

Kernel fusion has a shortcoming as well. The main problem is that it cannot vary the proportion of the mixing. Because the kernels are physically compiled together, and the new kernel has a fixed number of workgroups, there is no room for controlling the combination ratio adaptively.

In this thesis, we propose another approach to construct concurrent kernels. It adopts the source-to-source kernel transformation proposed in kernel fusion but improves it by enabling finer control on the mixing ratio. We will show more details about this approach in the following subsections.

5.3.1 Overview of Kernel Merging

We create a concurrent kernel by a source-to-source transformation with candidate kernels as components of the combined one. The newly created kernel conditionally executes either of the original ones. Depending on a condition, we can vary the proportion of each original kernel executed.

Listing 5.1 shows a simple example of merging two kernels together. The arguments belonging to the distinct kernels chain up to form the new kernel's argument list. To make the merged kernel work properly, we have to take care of some semantic details. Since a combined kernel has a larger number of global threads than its components, the index referenced by calling the built-in function `get_global_id()` in the merged kernel is no longer valid. We need to map the thread index from the new global thread space back to its original thread space. We also have to take care of the NDRange of the combined kernel and its component kernels. Finally, we have to choose a proper condition to ensure the correct mixing proportion.

Listing 5.1: Example of two kernels merging

```
__kernel void merged_kernel(kernel-1 arguments, kernel-2 arguments){
    if(condition_is_true){
        source code of kernel-1
    }
    else{
        source code of kernel-2
    }
}
```

As shown in Listing 5.1, once two kernels are merged, the new kernel function takes arguments from both kernels as its argument. Besides, the new kernel has to acquire extra arguments about the number of global and the local threads needed by each of the kernels that are to be merged, so as to recalculate new thread IDs for each of them after merging. The thread ID recalculation method is presented in Listing 5.2. Once we use workgroup ID as a guide to have a finer control on the mixing ratio of two kernels, as shown in Listing 5.3, the thread index recalculated by the method in Listing 5.2 would be spoiled, as neither of the sub-kernel has continuous workgroups. Therefore, instead of recalculating thread index directly from the API function `get_global_id(0)`, we firstly transform the workgroup index from discrete to continuous, as shown in Figure 5.2. Then, we use the new workgroup ID to calculate

thread index for each sub-kernels.

5.3.2 Thread Index Transformation

Hardware maps workgroups and threads directly to computing units. This one-to-one mapping simplifies the design complexity by using logical objects as physical ones and works efficiently when this scheme is dealing with a single kernel. However, when kernels contain separate logical functions, this one-to-one mapping fails. We need to decouple logical workgroups and threads from their physical equivalents.

We adopt the approach proposed in Elastic-Kernel [Pai et al. (2013)] to transform threads from logical space to physical space. The key idea is to attach extra parameters about each sub-kernel's global NDRange and workgroup information to enable recalculation of thread identifiers. The recalculation takes place in every sub-kernel separately. Listing 5.2 shows the details of the threads recalculation.

Listing 5.2: Pseudo code for thread index transformation. `global_size_0`, `global_size_1`, `local_size_0`, and `local_size_1` are the arguments passed to the merged kernel that represent the original kernel's NDRange and workgroup information. `block_id` and `thread_id` are logical workgroups and thread identifiers. `block_index_x` and `block_index_y` indicate the index of the workgroup in `x` and `y` dimension. `thread_index_x` and `thread_index_y` designate the offset of a thread within the workgroup in `x` and `y` dimension.

```
int gtid = get_global_id(0)
if(gtid < global_size_0 * global_size_1){
    int block_id = gtid / (local_size_0 * local_size_1);
    int thread_id = gtid % (local_size_0 * local_size_1);
    int block_index_x = block_id % (global_size_0 / local_size_0);
    int block_index_y = block_id / (global_size_0 / local_size_0);
    int thread_index_x = thread_id % local_size_0;
    int thread_index_y = thread_id / local_size_0;

    source code of original kernel follows here
}
```

We have a unique thread identifier by calling the OpenCL built-in function `get_global_id(0)`. As long as the thread identifier obtained by the API function falls into the range of logical space, it is a valid index; otherwise, the index goes beyond the bound-

ary and does not belong to the related original kernel.

We then easily identify which logical workgroup it belongs to by dividing it by the original workgroup size. Subsequently, we can also obtain the index on each dimension of both workgroups and local threads. By replacing each thread or workgroup ID with the transformed logical one, the sub-kernel can be executed as before, no matter how many other kernels co-exist.

5.3.3 Kernel Mixing Ratio

With the support of the thread transformation above, we can mix kernels. As shown in listing 5.1, the merged kernel is created by putting the original kernels into separate branches of an if-statement. The condition is responsible for the sub-kernel selection and controls the kernel mixing ratio.

Listing 5.3: Pseudo code for merged kernel

```
__kernel void opencl_kernel(kernel-1 arguments, kernel-2 arguments,
       const int kernel1_global_size_0, const int kernel1_global_size_1,
       const int kernel1_local_size_0, const int kernel1_local_size_1,
       const int kernel2_global_size_0, const int kernel2_global_size_1,
       const int kernel2_local_size_0, const int kernel2_local_size_1,
       const int M, const int N){
   if(get_group_id(0) % M < N){ // Launch kernel 1

       // recalculated thread id
       ... ...
       // original kernel-1
       ... ...
   }
   else{  // Launch kernel 2

       // recalculated thread id
       ... ...
       // original kernel-2
       ... ...
   }
}
```

In the OpenCL standard, the workgroup is the core scheduling unit. Each workgroup is assigned to a computing unit. Once a workgroup is allocated, it cannot migrate from one computing unit to another. Threads within the same workgroup have private register files but can share a common cache. The workgroup is allocated to a computing unit until all of its consisting threads are finished. This characteristic of the workgroup makes it a good guide to control the kernel mixing. The reason for controlling kernel mixing at workgroup level, instead of thread level, is that no extra divergence will be introduced by this method.

Listing 5.3 gives an example of how to use workgroup control kernel mixing ratio. There are two extra parameters, namely `M` and `N`, where `M` is the number of computing units available, and `N` is the number of units `kernel1` is to run. For example, if there are 16 computing units in a particular GPU, then at any given time 16 workgroups could run in parallel. So, we can set `M` to 16 since it represents the maximum number of parallel running workgroups. N could be any number between 0 and 16. Then, when a workgroup arrives, the hardware selects and performs the corresponding part of the code within it by comparing its ID with `M` and `N`.

Although by using workgroup ID, we can control the kernel mixing ratio, this introduces another problem that splits the thread space into two parts. Each sub-kernel holds only a part of the space and the other sub-kernel has the rest. The space splitting is caused by division operation on workgroup ID. Therefore, the workgroups in neither sub-kernel have a continuous index and the thread ID acquired by calling `get_global_id(0)` is not continuous as well. Noncontinuous thread index invalidates the original kernel semantics. In the next section, we show how to tackle this problem by using workgroup ID transformation.

5.3.4 Workgroup ID Transformation

Workgroup based selection causes a problem of thread space splitting. Take two kernels 2:3 mixing for example as shown in figure 5.2. When we launch it, we would like to have 40% of the hardware computing units serve one component kernel, and let the other one have the remaining 60%. Here we set M and N to 5 and 2 to get the 2:3 mixing ratio. For the workgroups if its remainder for modulo operation is less than 2, it will perform its sub-kernel1 part of the code; otherwise, it will activate its sub-kernel2 functions.

The left-hand side figure in figure 5.2 shows the example. In this example, the

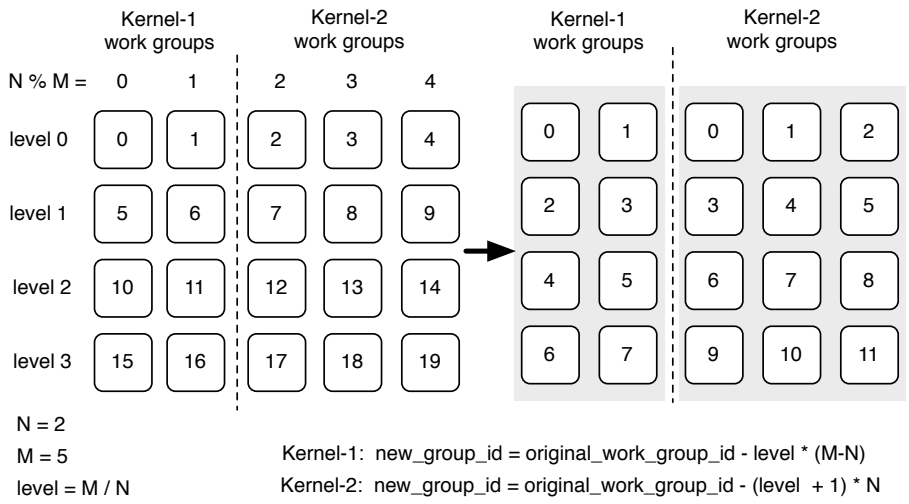


Figure 5.2: Discrete to contiguous conversion

compound kernel has 20 workgroups; each workgroup contains the code for both original `kernel-1` and `kernel-2`. After modulo operations on ID, those workgroups have an identifier of 0, 1, 5, 6, 10, 11, 15, and 16 will choose to run `kernel-1`. The others that are holding an ID of 2, 3, 4, 7, 8, 9, 12, 13, 14, 17, 18 and 19 will perform `kernel-2` instead. Consequently, though sub-kernels have enough workgroups to fulfill their function, the identifiers of these workgroups are not continuous, which lead thread identifiers among these workgroups to be no longer continuous either.

The irregularity in workgroup IDs is caused by splitting them into two parts, according to their physical ID, to perform different sub-kernels. Take kernel-1 in Figure 5.2 for example. Its third workgroup has a physical ID of 5, but logically it should have a value of 2. We format the workgroups into a 2D matrix with each row has `M` workgroups. Since the kernel selection has been made by a physical workgroup ID modulo `M`, the workgroups locate at the left-hand side of the dotted line belongs to kernel-1 and the ones at the right-hand side belongs to kernel-2.

The fundamental idea is to skip the unrelated workgroup IDs that belong to the other kernel on the same row (or level). Therefore, for the sub-kernel that has physical workgroups start from 0, there is no shift for workgroups on `level-0`. On `level-2`, the workgroups have to move forward by `M-N` steps because those `M-N` workgroups are taken by the other sub-kernel physically. Next, for the workgroups on `level-3`, they are asked to move `2*(M-N)` steps. `M-N` steps for the places in `level-1` and another `M-N` steps for the holes in level-0.

The logical workgroup IDs are recomputed by using equation 5.2.

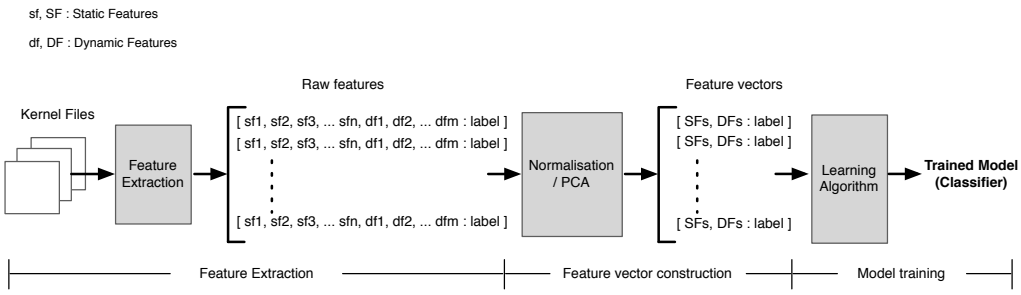


Figure 5.3: Training a predictive model. Raw features are firstly extracted from kernel files, which contain static features, and runtime parameters, or dynamic features. The feature vectors normalized and reduced PCA. They are summed from two kernels according to their merging proportion. Finally, the learning algorithm is trained on the feature vectors. As the result, the trained model will be used to classify the new coming kernels.

$$Kernel1 : new\_group\_id = physical\_group\_id - level * (M - N) \tag{5.1}$$

$$Kernel2 : new\_group\_id = physical\_group\_id - (level + 1) * N \tag{5.2}$$

After transforming the logical workgroup IDs back to continuous integer sequence, the global thread indices for each sub-kernels can be recalculated by using equation 5.3. Since the physical workgroup size is fixed, we can use it with the transformed workgroup indices to get the logical global thread indices for the sub-kernels:

$$new\_global\_id = logical\_group\_id * get\_local\_size(0) + get\_local\_id(0) \tag{5.3}$$

5.4 Predictive Model

Rather than pairing up kernels according to their profiling information, this chapter proposes a machine learning approach to find out the appropriate kernels that can benefit from concurrent execution on the same GPU processor. This section describes how to build the model with the kernel's static features and runtime parameters as input.

5.4.1 Overview

The predictive model is trained off-line by using a supervised learning algorithm. The training data consists of a pair of feature (the input object) and label (the desired output

value). Fed by the training samples, the learning algorithm deduces a function to map the features to the related labels. This function works as a pre-trained model on the unseen instances to label them into various categories. The learned model classifies the new data into a category.

The model in this chapter is trained from program features, which include both static and dynamic features. Figure 5.3 shows the training process, which is further divided into three sub-steps. Firstly, the raw features are derived from the application. As OpenCL kernels are just-in-time (JIT) compiled, the code features, namely static features, are extracted by a JIT compiler. Runtime parameters, such as the input sizes, the number of working threads, and the NDRange of a kernel instance, are acquired after the application started but before its kernel is launched. Once the raw features are obtained, they are processed to construct feature vectors. Several procedures have taken place in this step, such as features normalization, feature space reduction, and concurrent kernel features creation. Finally, a learning algorithm works on the feature vectors to train the model that will be used to classify unseen kernels according to whether they can experience performance improvement from concurrent execution, or not.

In the following section, we first introduce the machine learning algorithm, then describe how features are obtained and manipulated to create feature vectors. Finally, a graph based scheduling approach is used to find out the maximum pairs of concurrent kernels.

5.4.2 Machine Learning Model

The predictive model trained in this chapter is based on Support Vector Machine (SVM) and used to decide whether or not to run two kernels concurrently. The SVM classifier, with a radial basis function (RBF) kernel, performs a non-linear classification that maps the features into a high-dimensional space, then constructs a hyperplane in this high-dimensional space to do the classification.

As a supervised learning model, SVM is trained from a group of training samples that consist of features and labels. The features describe the kernel codes static characteristics and dynamic parameters. As OpenCL kernels are JIT compiled, the static features are extracted by a JIT compiler when it is called by the application at the runtime. Dynamic features can be derived before kernel launch, and they include the input/output buffer size, the number of working threads, and the NDRange of the

kernel instance. The labels in each training sample represent whether or not the kernel pair in that sample could experience performance improvement from concurrent execution. For training samples, we exhaustively run the kernels in pairs and compare the execution time with the sequential processing time so as to get the performance label. As this procedure is done to train the model, the overhead is a one-time cost and after the model has been built no further overhead will be introduced.

5.4.3 Task Features

The features used in training the model carry the critical information about the application. As a kernel's performance is determined by its function and the working environment, therefore, two types of features that are used in this chapter, which are static features and dynamic features.

Platform information is excluded from the training features to limit the feature space. Details about the hardware resources, such as the number of processing cores, processor/memory working frequencies, the number of cache layers, and cache sizes are not included as training features, as all these configurations are fixed for any particular platform. We, therefore, build a model per platform. The impact of hardware settings affects the application in the way of its execution. Hence, the characteristics of a program and its performance on a particular platform carry all the information a predictive model needs for classification.

Static Code Features

The static code features of a kernel is a set of integer values that describe the number of instructions of each instruction type. The instructions are derived from the intermediate representation (IR) that is generated by a JIT compiler. Code features carry information about the kernel, such as the number of operations in computing, memory accessing, and branches choosing.

Instructions of different types have separate latencies and throughput. We, therefore, weighted them differently in static features. Table 5.2 shows the details of the weighted instructions for both Nvidia and AMD platform. We use the time of integer addition as a scale to measure other operations. By assuming integer addition cost 1 unit of execution time, other operations could be calculated accordingly. Taking Nvidia's Kepler GPU for example, each streaming multi-processor has 192 single-precision CUDA cores, which could perform integer or float operations. Therefore,

Table 5.2: Weights of static features. Different operations performance are measured in the weight of interger operations. For Nvidia GTX 780, the performance of double precision is 1/24 of single precision. Hence, the double precision operation's weight is scaled up by 24 over single precision operation. Similar calculations take place on AMD double precision operations as well.

	NVIDIA (GTX 780)		AMD (Radeon 7970)	
Operation	Type	Weight	Type	Weight
Add, Sub, Max, Min	int	1	int	1
	float	1	float	1
	double	1×24	double	1×4
Mul	int	1	int	1
	float	1	float	1
	double	1×24	double	1×4
Mad	int	4	int	4
	float	4	float	4
	double	4×24	double	4×4
Div	int	8	int	8
	float	8	float	8
	double	8×24	double	8×4
And, Or, Xor, Shl, Shr	int	1	int	1
	float	-	float	-
	double	-	double	-
sinf, cosf tanf, expf sqrt	float	6	float	6
Load, Store	-	100	-	100

single-precision and integer operation have the same weight. Double-precision in this particular Kepler GPU is 1/24 of single-precision performance; therefore, its operation is weighted 24 times of the basic unit.

Multiply-and-Add (Mad) and Division are heavier than simple integer operation; therefore, they are highlighted by having a bigger weight. Logical operations in GPU processing cores usually cost one basic time unit (or clock cycle); hence, they have the smallest instruction weight. Special functions, such as sine and cosine, are performed on designated hardware, in Nvidia this hardware is Special Function Units (SFU). Typically, the number of SFU is far less than the CUDA cores on Nvidia platform, and the situation is the same for AMD GPUs. Therefore, the special operations have a higher weight.

Finally, memory accessing operations are given the heaviest weight, due to their long processing latencies. A load or store instruction could take hundreds of GPU clock cycles to prepare data to the processors or write results back to the global memory. The long latency of memory instructions is described with the highest weight in static features.

Runtime Parameters

Runtime parameters depend on input data. Typical dynamic features include the input/output data sizes, the NDRange sizes, the number of workgroups within the NDRange, and the number of working threads residing in the workgroup. Unlike static features, which are fixed and remain static for each kernel, dynamic features change with the input of the kernel. The size and the content of the input data, determine a kernel's dynamic features and impact the kernel's performance.

Feature Processing

Using raw features directly has two principal issues. First, the number of the features is large. There are around 50 kinds of instruction that could be derived from the LLVM intermediate representation. Too many features relative to the number of samples make model training difficult[Hawkins (2004), Ahmad and Narayanan (2015), Han (2014), Mierswa (2007)]. It may lead to overfitting where the model fits the training data too tight and performs poorly on the target instances. Second, the values of the feature have wide ranges, and this causes a problem for the learning algorithm, as the feature with a broad value range can dominate training. In order to tackle the above problem, features

Table 5.3: Combined feature vector for concurrent kernel model

	Features	Description
F1	comptInst / (allInst)	compute instruction ratio
F2	memInst / (allInst)	memory instruction ratio
F3	br / (allInst)	branch ratio
F4	barriers / (allInst)	barriers ratio
F5	dataSize / (MaxMem)	datasize
F6	dataSizeRatio1	first kernel datasize ratio
F7	dataSizeRatio2	second kernel datasize ratio
F8	globalWorkSize1 / (globalWorkSize)	first kernel global threads ratio
F9	globalWorkSize2 / (globalWorkSize)	second kernel global threads ratio
F10	localWorkSize1 / (localWorkSize)	first kernel local threads ratio
F11	localWorkSize2 / (localWorkSize)	second kernel local threads ratio

are grouped together according to their similarity and normalized into a unified range before use. Finally, we combine the features from two candidate kernels to create the feature vectors, with which the machine learning algorithm learns and evaluate the performance of two kernels concurrent execution. The combined features are shown in table 5.3.

5.4.4 Kernel Scheduling

The new arriving kernels, are classified by the off-line trained model in pairs according to the estimated performance categories, which identify either good or poor throughput compared to sequential execution. In practice, there are multiple pairing up schemes for a given kernel. The number of the kernels that can co-run with a given kernel and improve the throughput could be 0, 1, or any the number between 1 and n-1 (here n stands for the whole number of tasks). When the pairing choice is 0, it means that the kernel is better to run alone on the GPU because the system throughput would decrease when it co-runs with any other n-1 kernels. The existence of multiple pairwise schemes requires a proper scheduling method to maximize the number of concurrent kernel pairs.

Kernel Pairs Selection

Kernel pairs selection has a great impact on overall performance. To discuss the effect of various pairing schemes, consider the example in figure 5.4. There are five kernels, which are represented as `a-f`, queuing for the GPU device. In figure 5.4, we use value `1` and `0` to identify the associativity of two kernels. The value `1` represents that running

the two selected kernels in parallel is better than in sequential, and the value `0` means the opposite.

As the scheduler is only interested in the kernel pairs that could improve performance, we use a graph $G = (V, E)$ to describe the relationship of the kernels. The vertices represent the kernels and the edges between two vertices means running those two kernels concurrently on GPU has a better overall performance than executing them sequentially. Kernels that are not suited for co-running are disconnected from each other in the graph. Figure 5.5a shows the pairwise kernel graph for the example in figure 5.4.

The degree of the vertex in the pairwise kernel graph identifies the number of other kernels that could experience throughput improvement when they pair up with the kernel that is represented by the vertex. In figure 5.5a, the kernel `a` has the best associativity. The system has an optimized throughput when kernel `a` pairs up with any of the four kernels from `b`, `d`, `e`, `f`. Vertices, like `c` and `e`, have poor associativity, they can only improve the performance when space sharing the GPU with particular kernels, in this example they are vertices `b` and `a`.

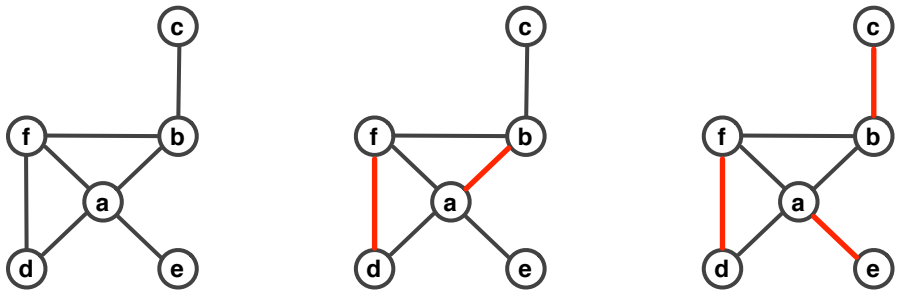
Selection algorithms schemes affect the performance by selecting a different number of kernel pairs. Figure 5.5b and figure 5.5c show two selection methods. In figure 5.5b, two pairs of kernels, which are `<a,b>` and `<d,f>` are selected, and the rest of the kernels (`c` and `e` in this case) are running sequentially since co-running them would lead to a decreased throughput. Figure 5.5c shows a better pairing scheme, in which all kernels are paired up, and it has an optimized performance compare to the scheme in figure 5.5b.

Scheduling Algorithm

Finding the maximum number of kernel pairs is equivalent to finding out the maximum matching in graph theory. In graph theory, a matching is described as a set of edges, in which no vertex is shared by two edges[Savage (2016)]. Therefore, the problem of finding the maximum kernel pairs can be equally transformed into the problem of finding out the maximum independent set in a graph. In this thesis, we use blossom algorithm [Edmonds (1965b), Edmonds (1965a), Ahn et al. (2015), Blum (2015), Kavathekar (2005)] to find the maximum matching. It is a well-known algorithm as it has a polynomial complexity. The key idea of the blossom algorithm is to find the odd-length cycle in the graph, namely blossom, and then contract it into a single vertex. After that, there are no cycles in the transformed graph and the independent edges can

	a	b	c	d	e	f
a	0	1	0	1	1	1
b	1	0	1	0	0	1
c	0	1	0	0	0	0
d	1	0	0	0	0	1
e	1	0	0	0	0	0
f	1	1	0	1	0	0

Figure 5.4: Five kernels (`a` to `f`) pairing associativity. If two kernels co-execution is faster than running them sequentially, this pair of kernel is identified by `1`, otherwise by `0`.



(a) Kernel Pairing Graph (b) Sub-optimal Pairing (c) Best Pairing

Figure 5.5: The kernel pairs can be represented as a graph, in which a kernel is represented as a vertex and the edge between two vertices means the co-execution of corresponding two kernels can improve performance. A greedy pair selection scheme pairs up whatever the kernels it encounters first. In many cases, this leads to a sum-optimal selection (shown in b). The best selection can find the maximum number of kernel pairs and outperform the greedy method.

be calculated. As the blossom is the odd-length cycle, the independent edges can be easily calculated.

Listing 5.4 shows the algorithm. Each vertex is labeled as odd or even alternatively by traversing the graph. The starting vertex is labeled as even, and the next one is labeled as odd, and so on. Next, the algorithm shrinks the graph by contracting the blossom into a vertex. Once all blossoms are replaced by vertices, the independent edges are calculated.

Listing 5.4: Maximum Matching by Blossom-Algorithm. Taken from [Kavathekar (2005)]

```
Init an empty matching, M
while(a blossom || an augmenting path){
    Grow forest, labelling the vertices "even" and "odd"
    if(there is a blossom in the graph)
        shrink the blossom to obtain a new graph G
        continue
    else
        find all even-even edges to obtain a maximally disjoint set of
            augmenting paths
}
```

5.5 Experiment Setup

We compare our approach to two existing schemes.

Elastic Kernels (EK) This approach runs two kernels concurrently and merges the corresponding host programs. It does not have a model to determine what to merge. It co-executes all kernels in pairs.

Energy-Efficient Concurrent Kernels (EECK) This approach is similar to EK but requires profiling of the applications beforehand to determine what to merge. It uses profiling information from a small data set to guide kernel mergings for the larger data set.

Separate or Concurrent on GPU(SoC_GPU) Our approach to concurrent execution of kernels.

To make a fair comparison, we implement EK and EECK and ignore the introduced overhead *i.e.* the cost of profiling and recompilation. Such overheads usually outweighed the benefits, making them hard to use in practice. Throughout the com-

Table 5.4: Hardware platform

	Intel CPU	**NVIDIA GPU**	**AMD GPU**
Model	Core i7 4770K	GeForce GTX 780	Radeon HD7970
Architecture	Haswell-DT	Kepler GK110	Tahiti XT
Core Clock	3.4 GHz	1215 MHz	1000 MHz
Core Count	4 (8 w/ HT)	2304	2048
Memory	16 GB	3 GB	3GB
Memory Bandwidth	21GB	288 GB	264 GB

Table 5.5: Benchmarks

Suite	Benchmarks	Benchmark
Parboil	BFS	Mri-Q
	Sgemm	Spmv
	Sad	
Ploybench	ATAX	BICG
	CORRELATION	GESUMMV
	SYR2K	SYRK
	2DCONV	3DCONV
	GEMM	GRAMSCHMIDT
	2MM	3MM
	COVAR	FDTD-2D
	MVT	

parison, we use a unified metric, which is system throughput, to evaluate the results. The experiments are carried on a number of benchmarks from Parboil and Polybench benchmark suits.

5.5.1 Platform and Benchmarks

We evaluate on two CPU+GPU systems. Both have an Intel Core i7 4-core CPU and 16GB main memory. One platform contains an NVIDIA GeFore 780 and the other one contains an AMD HD 7970, see table 5.4. Both systems host OpenSUSE 12.3 Linux. We use LLVM 3.4 for JIT compilation and benchmarks are compiled using GCC 4.7.2 with -O3 option.

We restrict our attention to benchmarks with 1D and 2D NDranges from two mainstream OpenCL benchmark suites (see table 5.5): the Parboil and the Polybench benchmark suite giving 20 programs in all .

5.5.2 Performance Evaluation

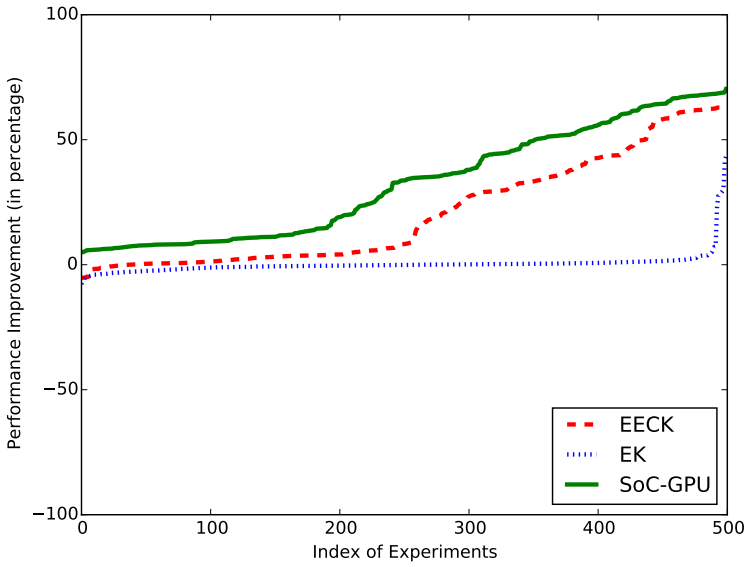
We evaluated our schemes with 500 different task configurations. We selected 10 different task queue sizes containing between 2 and 64 kernels. For each task queue size, we randomly selected 50 different programs, to give 500 configurations. As behaviour is dynamic, we evaluated each configuration 30 times and report the median performance. This results in 6000 experiments per policy. Performance is presented as speedup relative to executing in sequence on a GPU.

5.6 Results

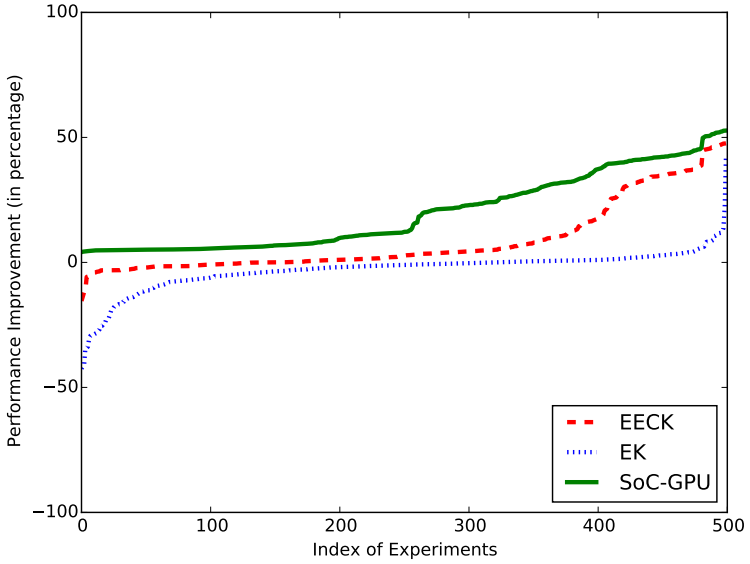
In this section, we evaluate our approach against alternative approaches and analyze the behavior and accuracy of our predictive model. As the performance is the main goal of runtime optimation and task scheduling, we focus our results discussion on system throughput in this and the subsequent chapter.

We divide the experiments into three groups according to the number of OpenCL kernels in the task queue. In the small group, there are less than 16 kernels; in the medium group, there are 16 to 32 kernels and in the large group, there are 32 to 64 kernels. Figure 5.7 shows the results of all three groups and the average performance improvement. The STP performance has been increased for all of the approaches on both of the platforms. The more kernels available means there are more opportunities for a successful concurrent execution. For EECK and our approach, there is a substantial performance improvement with the number of kernels increasing. The gain is larger on NVIDIA probably due to greater spare GPU resources. However, the performance of EK is constantly poor, in many cases, it is even worse than the baseline. On average, our approach is 27% better than EK and 11% better than EECK on NVIDIA platform. On the AMD platform, our approach is 22.7% better than EK and 11.2% better than EECK.

Figure 5.6 shows more details of results on NVIDIA and AMD platform respectively. The x-axis is a ranked in order of experiment. On both platforms, our approach consistently outperforms the other techniques. In the worst case, our approach is 6% and 4% better than the baseline on NVIDIA and AMD platform; however, EECK and EK are 4% and 5% worse than baseline on NVIDIA platform and are 10% and 45% worse than baseline on AMD platform. Concurrent execution of the wrong kernels on AMD is costly.



(a) System throughput improvement on NVIDIA platform



(b) System throughput improvement on AMD platform

Figure 5.6: Performance of all the GPU only experiments on NVIDIA and AMD platforms. In 500 different task configurations of queue contains 2 to 64 kernels, our approach consistently outperforms the other techniques. In the worst case, our approach is 6% and 4% better than the baseline on NVIDIA and AMD platform.

On maximum performance improvement, our approach is 70% and 52% better than baseline on NVIDIA and AMD platform, while the corresponding results for EECK and EK are 63%, 41% on NVIDIA platform, and 46%, 31% on AMD platform.

5.7 Analysis

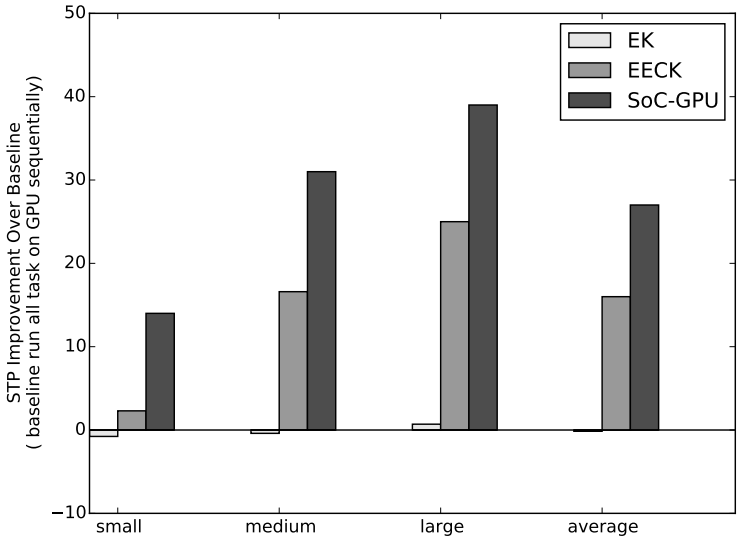
When running two kernels concurrently, the characteristics of these kernels have an impact on the resulting performance. Such characteristics includes computation and memory access intensity, branches within each kernel, NDRanges, processing data sizes, and so forth. Here we examine the effects of kernel characteristics on performance to see if it can provide insight into constructing concurrent kernels. To keep the length of this chapter short, all results in the following sections are from the Nvidia platform.

5.7.1 Impact of computation intensity

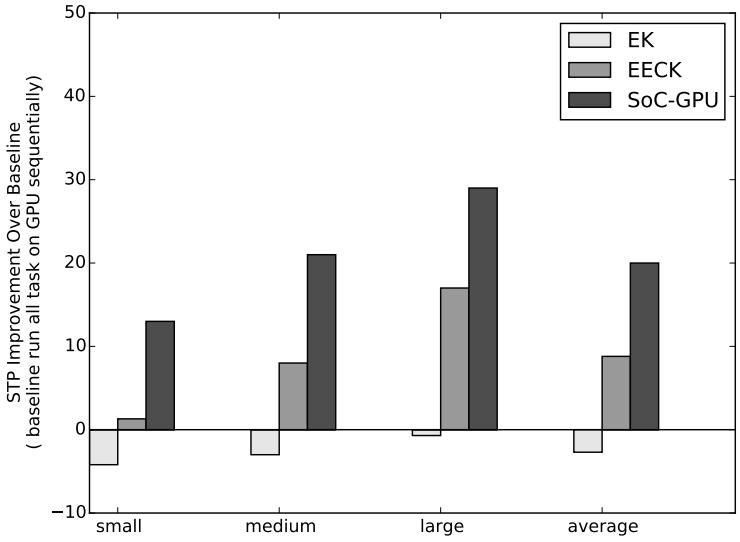
Figure 5.8-a shows the impact of kernel compute intensity on two kernels concurrent execution. The candidate kernels are listed on the x-axis and y-axis separately. The dot in the graph represents the performance of concurrent execution of two associated kernels, one from the x-axis and the other from the y-axis. The color of the dot represents whether the concurrent execution improves performance, or not. If the dot is blue, we can expect a better throughput from the two corresponding kernels co-running. Whereas, if the dot is red, there is a performance slowdown introduced by concurrent execution. The size of the dot shows the changes on performance in either direction. A bigger blue dot means better improvement and a bigger red dot stands for worse slowdown.

The figure is diagonally symmetrical because the same kernels are listed on the x-axis and the y-axis. We sort these kernels according to their computing intensity. On the x-axis, the kernel located on the left side has a less computing intensity, and on the y-axis, the kernel located higher are more compute-bound. So, a dot near the left bottom corner represents the performance of two less compute-bound kernels performance. A dot locate near the upper top corner represents two heavy compute-bound kernels co-execution.

As we can see, merging two compute-intensive kernels is normally a bad idea, as in most of the cases the system suffers a slowdown from it. On the other hand, merging



(a) System throughput improvement on NVIDIA platform



(b) System throughput improvement on AMD platform

Figure 5.7: System throughput improvement for concurrent kernel execution. On average, our approach is 11% better than EECK and 27% better than EK on NVIDIA platform; 11.2% better than EECK and 22.7% better than EK on AMD platform.

two less compute-bound kernels not necessarily leads to performance improvement. As we can see in the figure, half of them experience a higher throughput and results of the other half are opposite. The performance varies a lot when a compute-intense kernel runs with another less compute-intensive kernel. It depends on the specific kernel pairs.

Figure 5.8-b summarizes the result by showing the percentage of kernel pairs that have optimized throughput via concurrent execution. We show the result in three categories. The bar in each category represents the number of concurrent kernels that have a throughput improvement versus the overall kernel pairs in that category. The first category represents two low compute-intensive kernels co-running. The second category stands for one high compute-intensive kernel and one low compute-intensive kernel co-executing. The last category is two high compute-intensive kernels concurrent executing. As we can see, when we run two low compute-intensity kernels, 60% can benefit from their co-execution. When it comes to the combination of a high compute-intensity and a low compute-intensity kernel, the probability of having a performance gain from it drops to 50%. When we run two high compute-intensive kernels, there is only 27% that we can have a benefit from kernels co-running.

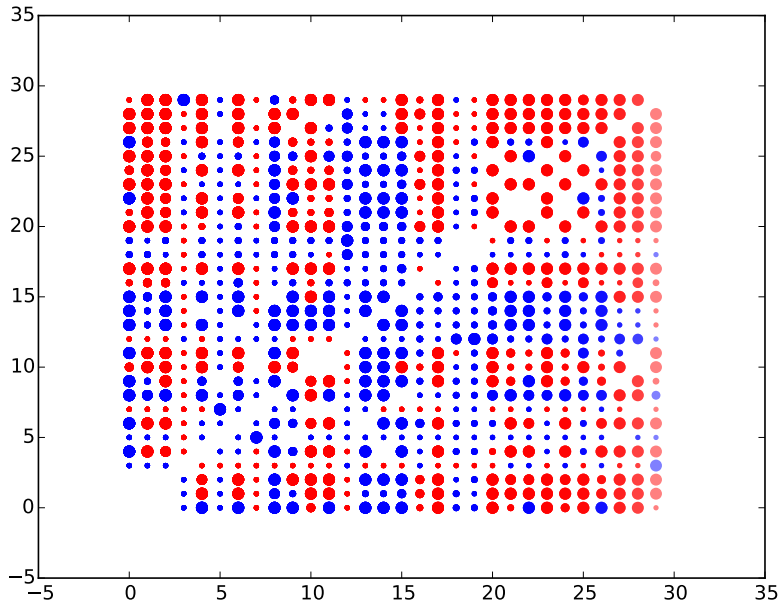
Figure 5.9 presents the result of kernel co-running in all three categories that have separate mixing schemes. On average, only two low computing-intensive kernels co-execution improves the throughput by 7%. Co-running of low intensive and high intensive kernels suffers a slowdown of 35%. The slowdown increased to 116% when two high compute-intensity kernels are running in parallel.

5.7.2 Impact of memory intensity

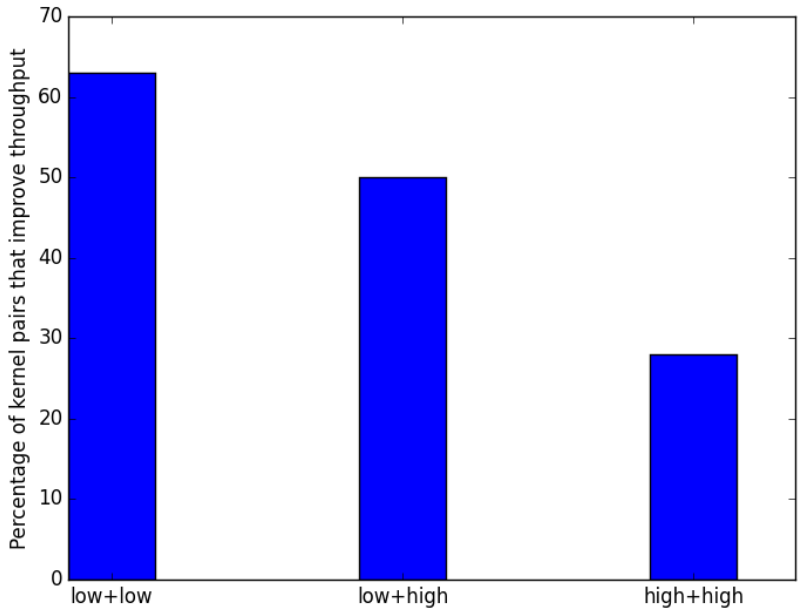
Figure 5.10-a shows the impact of the memory intensity. As in the case of computing intensity, the blue dots represent performance improvement, and red dots stands for a slowdown. The kernels on x-axis and y-axis are aligned according to their intensity.

From the figure, there is no clear pattern to identify which pair of kernels concurrent execution could have a performance improvement or a slowdown.

Figure 5.10-b shows the summary result. As before, the kernels are divided into three categories. In the first category, two kernels with low memory accessing intensity run concurrently. 65% of kernel pairs in this category can improve throughput. When the two kernels, one has low memory intensity, and the other has high memory intensity, they fall into the second category. In this group, 50% of the kernel pairs can



(a) Performance distribution when aligning kernels along their computation intensities. Kernels are aligned in X-axis and Y-axis ascendingly, the bigger the index is the higher compute-intensity a kernel has. The performance of the concurrent kernel execution is designated by the size of the dot. Red dot means a slowdown and blue dot represent an improvement.



(b) The proportion of kernels that have a performance improvement in each category. When running two low-compute-intensity kernels in parallel, 63% of of the kernel combinations experience performance improvement. 50% and 30% of the kernels have improved throughput for low-high and high-high mixing correspondingly.

Figure 5.8: The impact of kernel computing intensity on constructing concurrent kernels

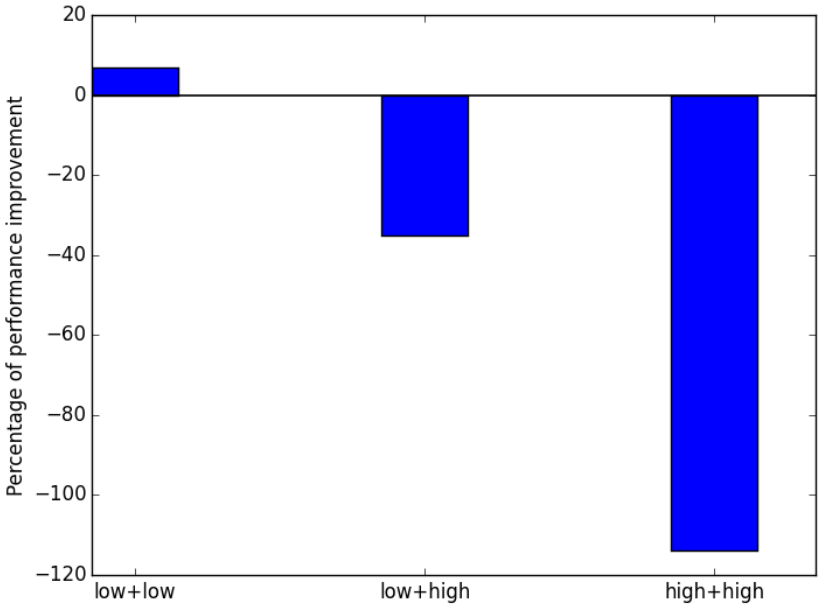


Figure 5.9: Performance summary of kernel co-running in all three compute-intensity categories.

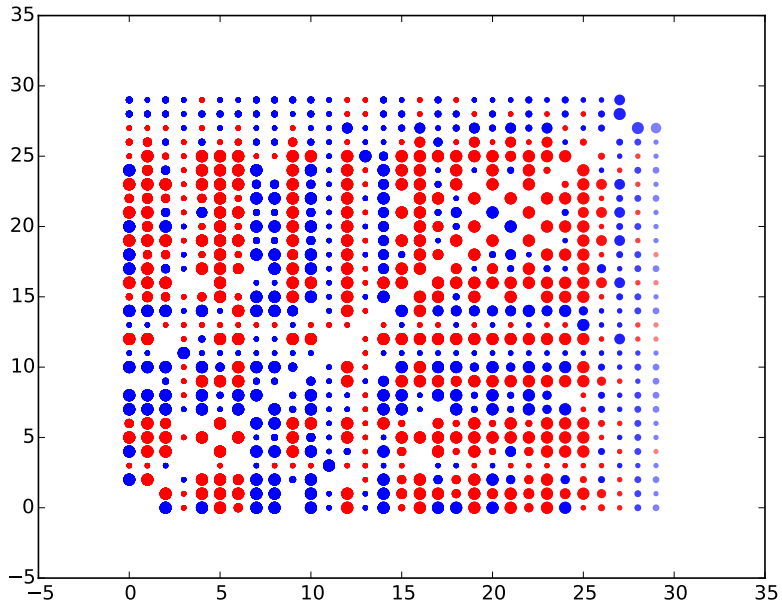
improve the system performance. Finally, when both of the kernels access memory heavily, running them together, 46% of such kernel pairs improve the performance.

Figure 5.11 presents the result of kernel co-running in all three categories that have separate mixing schemes. It shows that using memory intensities alone is a poor policy. On average, two low memory-accessing-intensive kernels co-execution decreases the throughput by 8%. Co-running of low intensive and high intensive kernels suffers a slowdown of 37%. The slowdown increased to 46% when two high computing-intensive kernels are running in parallel.

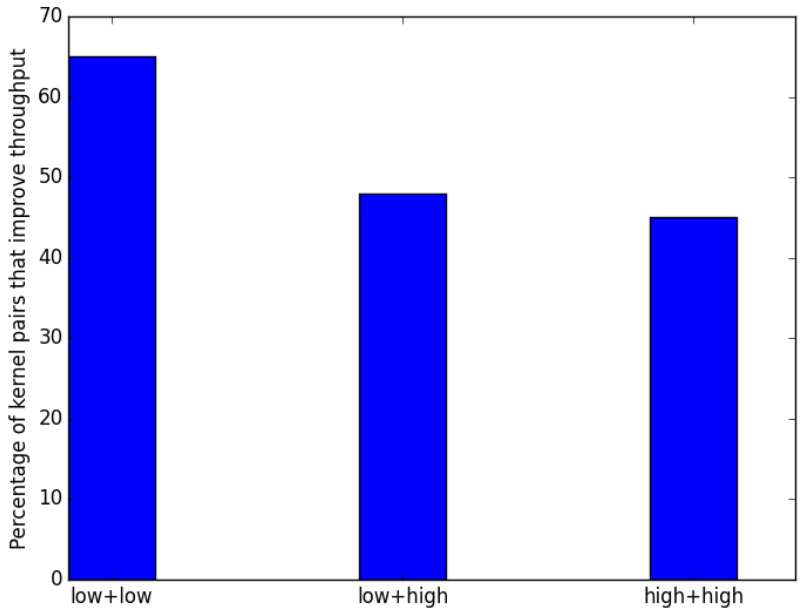
5.7.3 Impact of branches

Branches within the kernel have a big impact on a single kernel's performance. When kernel threads reach a branch, depending on the thread state and the branch condition, usually part of the threads can keep active. Therefore, branches within the kernel can hurt the parallelism. When running two kernels concurrently, the number of branches in each kernel could possibly affect the overall performance as well.

We examine the impact of branches in this section. Figure 5.12-a shows the result. Normally, concurrent execution of two kernels with fewer branches could improve the



(a) Performance distribution when aligning kernels along their memory accessing intensities. Kernels are aligned in X-axis and Y-axis ascendingly, the bigger the index is the higher memory accessing intensity a kernel has. The performance of the concurrent kernel execution is designated by the size of the dot. Red dot means a slowdown and blue dot represent an improvement.



(b) The proportion of kernels that have a performance improvement in each category. When running two low-memory-intensity kernels in parallel, 65% of of the kernel combinations experience performance improvement. 50% and 48% of the kernels have improved throughput for low-high and high-high mixing correspondingly.

Figure 5.10: The impact of kernel memory accessing intensity on constructing concurrent kernels

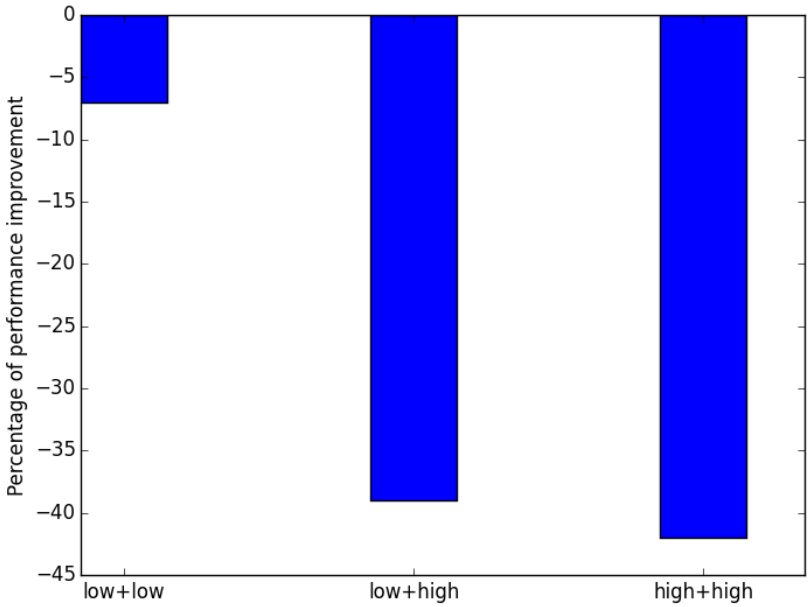
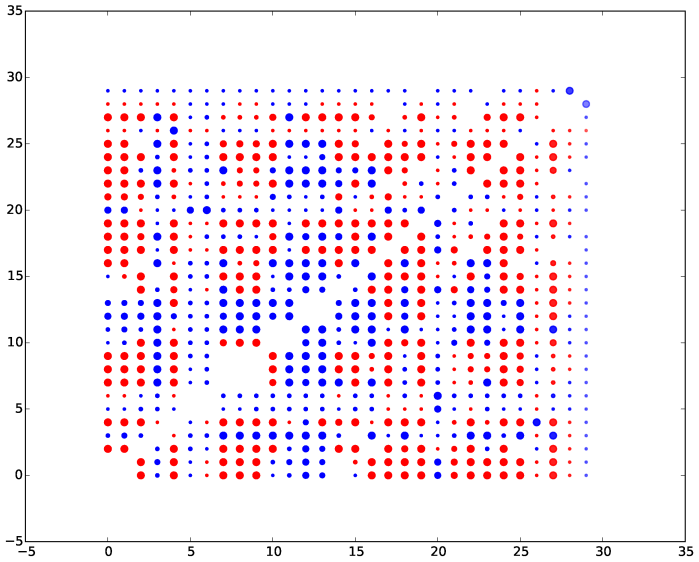


Figure 5.11: Performance summary of kernel co-running in all three memory-accessing-intensity categories.

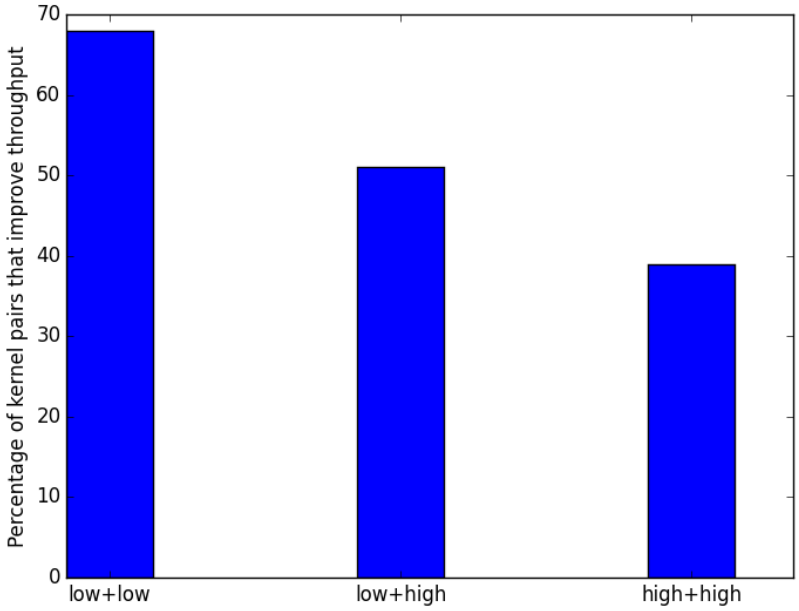
throughput. The co-execution of two kernels with more branches, on the other hand, has a higher chance to hurt the performance. When it comes to a combination, it highly depends on the kernels characteristics.

Figure 5.12-b shows the summary result. 65% of the concurrent kernels could improve the performance when both of the kernels have fewer branches. When of two kernels one has few branches but the other has many branches, 50% of such kernel pairs can have a good throughput when running concurrently. If both of the kernels have many branches in each of them, the average performance would drop. Only 38% of those kernel pairs achieve a good throughput.

Figure 5.13 presents the result of kernel co-running in all three categories that have separate mixing schemes. On average, with two low number of branch kernels co-execution degrades the throughput by 12%. Co-running of low and high branches kernels suffers a slowdown of 37%. The slowdown increased to 48% when two high branch kernels are running in parallel.



(a) Performance distribution when aligning kernels along their kernel branches. Kernels are aligned in X-axis and Y-axis ascendingly, the bigger the index is the more branches a kernel has. The performance of the concurrent kernel execution is designated by the size of the dot. Red dot means a slowdown and blue dot represent an improvement.



(b) The proportion of kernels that have a performance improvement in each category. When running two low branch-intensity kernels in parallel, 68% of of the kernel combinations experience performance improvement. 50% and 41% of the kernels have improved throughput for low-high and high-high mixing correspondingly.

Figure 5.12: The impact of kernel branches on constructing concurrent kernels

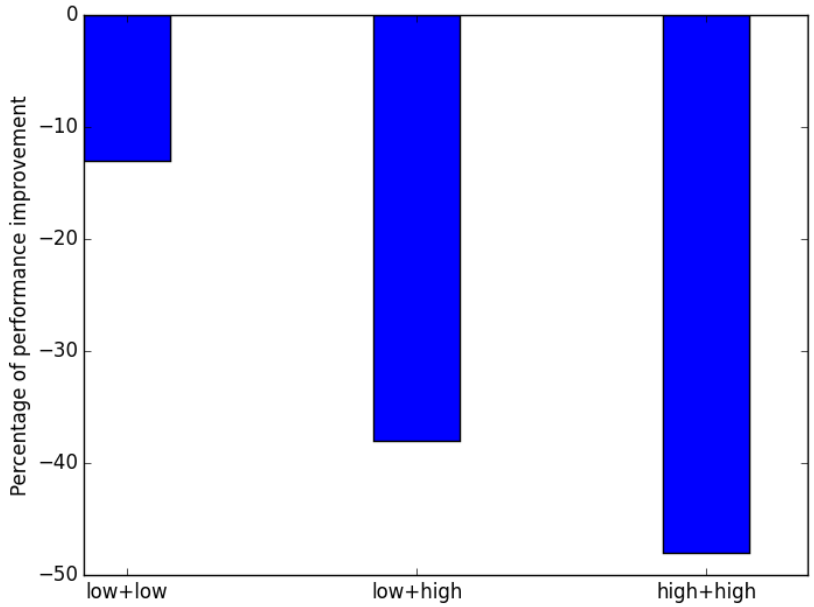


Figure 5.13: Performance summary of kernel co-running in all three branch-intensity categories.

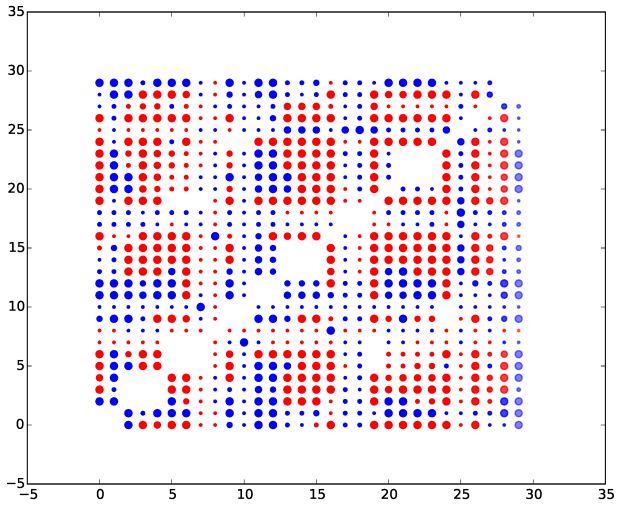
5.7.4 Impact of NDRange

NDRange has an impact on data locality, as threads belonging to the same workgroup could share the local memory. Therefore, the bigger the workgroup, the better performance would be. However, if threads within the same workgroup do not access memory in a coalesced way, they suffer a performance drop. Then, a bigger workgroup makes the performance worse.

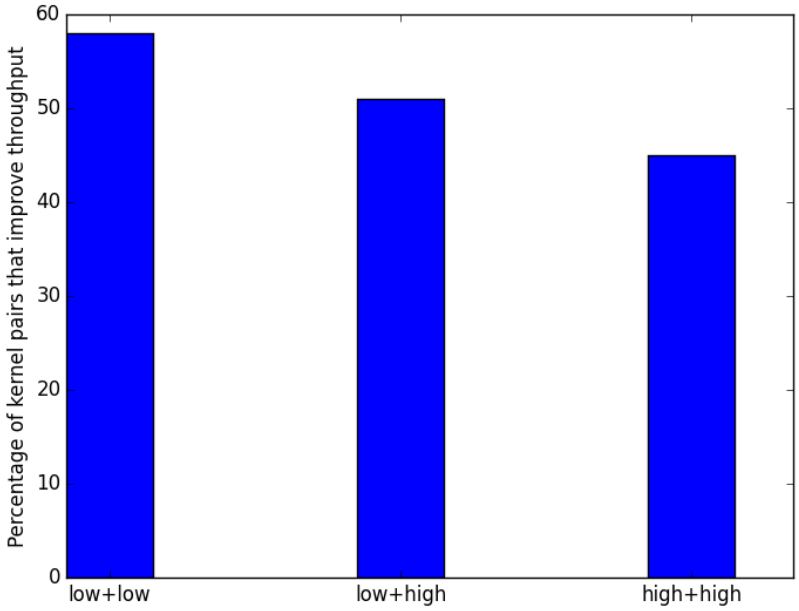
Figure 5.14-a shows the result of the impact of NDRange on concurrent kernel execution. When running two kernels, if both of them has 1D NDRange, they have a higher chance of improving performance. On the other hand, when both kernels have 2D NDRange, there is a lower probability of improving throughput.

Figure 5.14-b shows the summary of the result. 57% of the kernels pairs concurrent execution improves the system performance when both of the kernels have 1D NDRange. This number drops to 50% when one kernel has a 1D NDRange and the other kernel has a 2D NDRange. When both of the kernels have 2D NDRange, running them concurrently, only 45% of the total number of such kernel pairs can bring improvement.

Figure 5.15 presents the result of kernel co-execution in all three categories. On



(a) Performance distribution when aligning kernels along their kernel NDRange dimensions. Kernels are aligned in X-axis and Y-axis ascendingly, the bigger the index is the higher dimension a kernel has. The performance of the concurrent kernel execution is designated by the size of the dot. Red dot means a slowdown and blue dot represent an improvement.



(b) The proportion of kernels that have a performance improvement in each category. When running two low 1D NDRange kernels in parallel, 68% of of the kernel combinations experience performance improvement. 52% and 41% of the kernels have improved throughput for 1D-2D and 2D-2D mixing correspondingly.

Figure 5.14: The impact of NDRange on constructing concurrent kernels

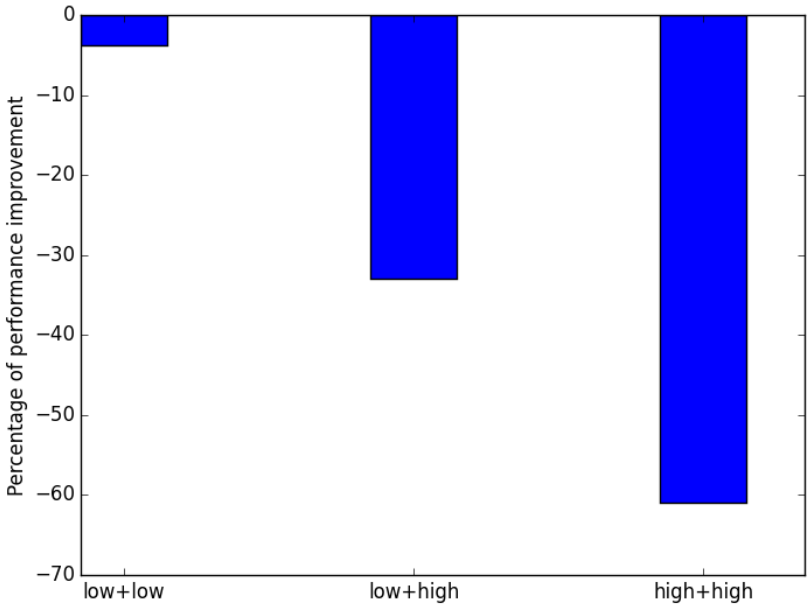


Figure 5.15: Performance summary of kernel co-running in all three NDRange-mixing categories.

average, only two 1D-NDRange kernels co-execution decreases the throughput by 4%. Co-running of 1D and 2D kernels suffers a slowdown of 32%. The slowdown increased to 59% when two 2D kernels are running in parallel.

5.7.5 Impact of data size

Finally, data sizes are also important to kernel performance. Larger size usually means more computations. It also means more time would be consumed by transferring data between CPU main memory and GPU memory. The overhead brought by data movement could outweigh the kernel execution in some cases. We analyze its impact when it comes to co-execution.

According to figure 5.16-a, two kernels co-execution would have poor performance when both of the kernels have small data sizes. When both of the candidate kernels have big data size, we have a higher chance of getting better performance. If one kernel has a big data size but the other kernel has a small one, the performance of concurrent execution is up to the kernels characteristics.

Figure 5.16-b shows the summary of the result. 18% of the kernel pairs could improve the overall performance when both of the candidate kernels have a small data

size. When the one has large data size and the other has small data size, 50% of such kernel pairs may have a good throughput. When both of the candidate kernels are dealing with big data size, running them concurrently has a higher chance, which is 67%, to end up with a good performance.

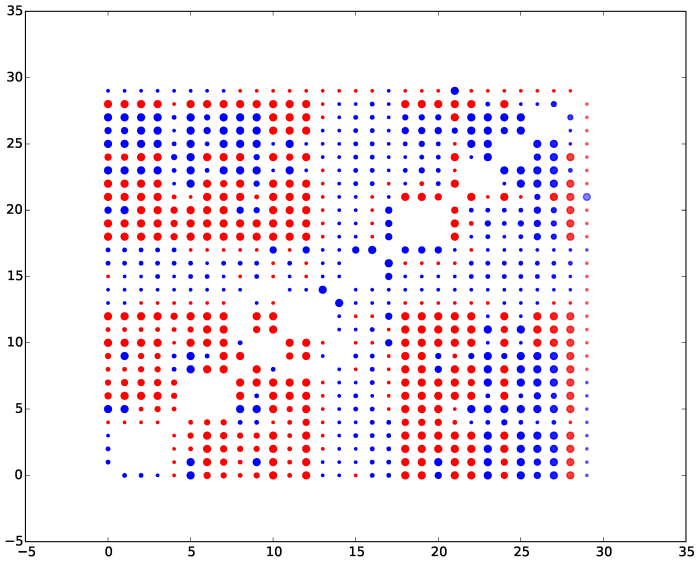
Figure 5.17 presents the result of kernel co-running in all three categories that have separate mixing schemes. On average, only two small datasize kernels co-execution improves the throughput by 77%. Co-running of small and large datasize kernels suffers a slowdown of 39%. The slowdown decreased to 4% when two large datasize kernels are running in parallel.

5.7.6 Summary of the analysis

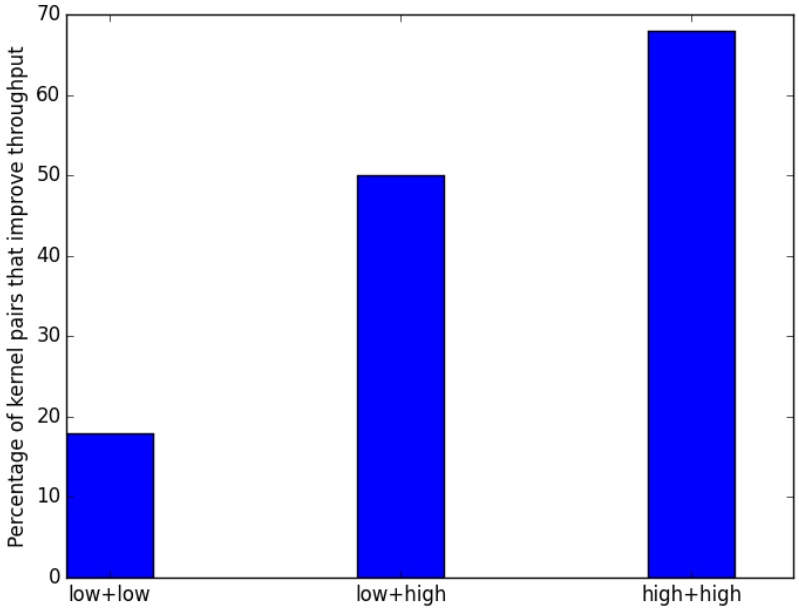
In this section we have analyzed the kernels co-execution under five separate schemes. The performance of pairwise kernel co-running is scattered when a single parameter is used to guide the kernel combination. On average, most of the co-running approaches decrease the system throughput except when running two low computing-intensive kernels in parallel. However, the performance improvement is trivial. Therefore, naive kernel combination guided by single parameter is not a promising approach in practice. To release a higher performance, the machine learning based approach is more attractive method because it is based on the knowledge that is learnt from a wide range of program features and runtime settings.

5.8 Summary

In this chapter, we described concurrent kernels execution on single GPU device to improve the throughput of the system. According to the experiment, running compute-intensive and memory-intensity kernels in parallel does not necessarily mean an optimized performance. In practice, kernels co-running is a complex problem that the kernel selecting method built on one or a few parameters can hardly be embraced by other kernels. Therefore, we develop a machine learning based approach to reveal the relationship between kernel's characteristics and the target devices. We evaluate our approach on two different platforms and have a superior performance comparing to other state-of-the-art methods.



(a) Performance distribution when aligning kernels along their input data sizes. Kernels are aligned in X-axis and Y-axis ascendingly, the bigger the index is the larger input data a kernel has. The performance of the concurrent kernel execution is designated by the size of the dot. Red dot means a slowdown and blue dot represent an improvement.



(b) The proportion of kernels that have a performance improvement in each category. When running two small datasize kernels in parallel, 58% of of the kernel combinations experience performance improvement. 52% and 43% of the kernels have improved throughput for small-large and large-large mixing correspondingly.

Figure 5.16: The impact of data size on constructing concurrent kernels

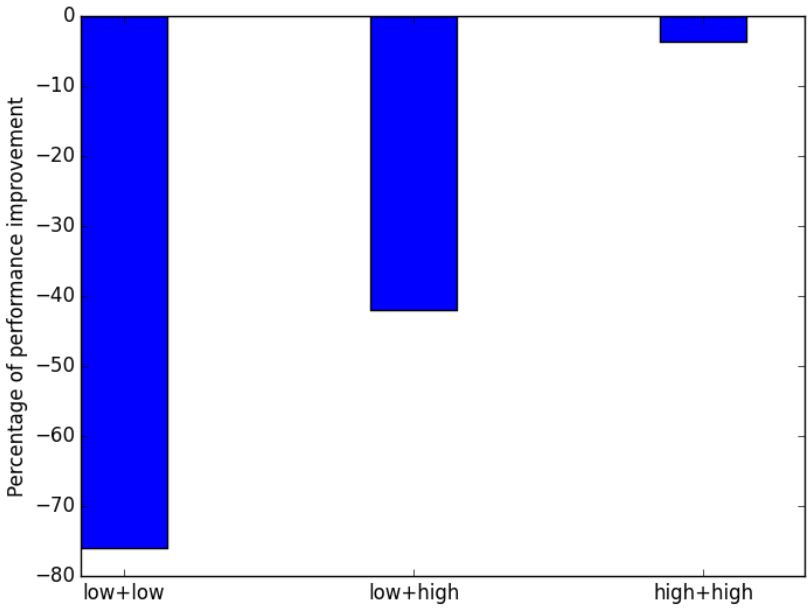


Figure 5.17: Performance summary of kernel co-running in all three datasize-mixing categories.

Chapter 6

Separate and Concurrent Kernel Scheduling

This chapter presents a runtime framework and a machine learning based scheduler that schedules single and concurrent kernels on CPU-GPU heterogeneous platforms.

This chapter is organized as follows: Section 6.1 introduces the background of single and concurrent kernel scheduling on CPU-GPU platforms. In this section, we have reviewed the related materials, which have been presented in Section 2 and 3, from the angle of a runtime framework. Section 6.2 presents the motivation of this chapter. Section 6.3 describes the overview of the runtime framework and the implementation details follow in Section 6.4. The experiment setup is presented in section 6.5. The experiment results are shown in section 6.6, and an analysis is presented in section 6.7. Finally, we summarise this chapter in section 6.8.

6.1 Introduction

Incorporating GPUs into multi-core parallel systems is increasingly popular. They provide the potential for high performance computing with relatively low power. Users typically write part of their applications as a kernel, using CUDA or OpenCL, which is then executed on a GPU

GPUs are normally used as dedicated accelerators for a single application. There is no overall operating system resource management, no hardware support for time sharing and very limited support for space sharing. This lack of support is a problem as GPUs become incorporated in mainstream parallel systems and used by multiple concurrent user applications [Margiolas and O'Boyle (2015)]. In addition, where ap-

propriate, we would like to share GPU resources amongst several jobs to fully utilize resources [Pai et al. (2013)].

While GPUs are excellent at accelerating certain jobs, they are poorly suited to others [Chapter 4]. Their suitability depends both on the job and the underlying GPU. Ideally, we would like to schedule jobs on the GPU only when appropriate and execute on the multi-core CPU host otherwise.

The research challenges are to determine (i) when to share the GPU among jobs and (ii) when to schedule jobs to the multi-core CPU. Given that this trade-off will vary based on programs and the underlying platform, we want an approach that is portable and low overhead. Furthermore, as we focus on a general purpose dynamic multi-user setting, we need an approach that does not require profiling or prior knowledge of the user program.

This problem of GPU utilization has been recognized by other researchers and there has been prior work in co-executing kernels on a GPU. However, such approaches are inappropriate for multi-user scheduling. Elastic Kernels [Pai et al. (2013)] (EK) is the best known work. Here, they statically merge the host code based on traces of the programs. This approach is unsuitable for multiple concurrent user jobs as it is unable to determine the best kernels to co-execute. In addition, the need to trace applications prior to execution in order to merge host code and recompile, is unrealistic in a live dynamic multi-user setting. In [Jiao et al. (2015)] they tackled the problem of determining the best combination of kernels by profiling every candidate application ahead of time and then selecting the two kernels that are likely to improve energy and throughput. While this approach may work in single-user cases where the same combinations of kernels are executed repeatedly, it cannot be used in a dynamic multi-user setting with unknown jobs where prior profiling is not feasible.

Both EK and Energy-Efficient Concurrent Kernel (EECK)[Jiao et al. (2015)] focus on sharing the GPU, however, they do not consider the host CPU as a potential scheduling target. As all GPU system systems have a host multicore, this is a wasted opportunity. In Chapter 5 we determine whether to schedule OpenCL jobs to a CPU or GPU. They show performance improvement over partitioning the job between CPU and GPU but do not consider scheduling multiple jobs concurrently to the GPU to exploit hardware resources.

This chapter develops a new scheduling approach for multiple OpenCL executing on CPU/GPU heterogeneous systems. It first determines which user jobs should be scheduled to the CPU and which to the GPU. It then determines if appropriate, which

kernels should be merged to improve performance. Merging of kernels is performed by a JIT compiler, while scheduling is performed by a thin runtime layer. Unlike previous approaches this is totally transparent to the user and requires no profiling or runtime overhead. Our approach relies on offline predictive modelling. It builds a one-off, statistical model "at the factory", based on offline experiments to determine the best merging and scheduling of kernels based on program features and runtime performance. This model is then cheaply deployed at runtime, taking program features as inputs and predicts the best scheduling decision as an output.

It is evaluated on a large number of workloads ranging from 2 to 64 jobs in size randomly selected from 20 benchmarks selected from the Parboil and Polybench benchmark suites. The framework was evaluated on the NVIDIA and AMD platforms where in each case we improve performance by more than 40% on average.

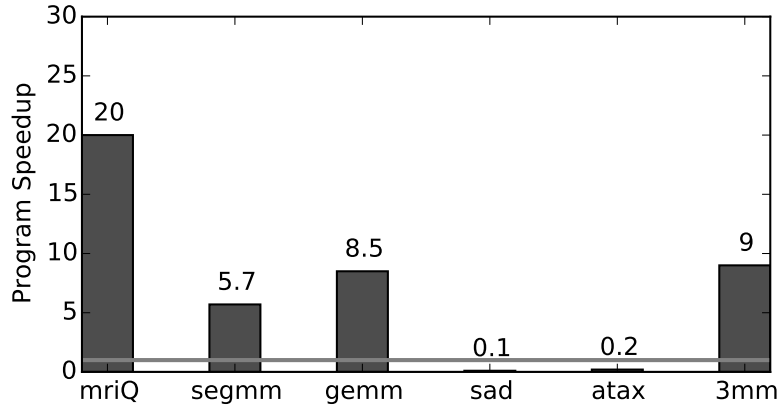
The next section provides a motivating example showing the need for accurate scheduling. This is followed by a description of our framework, the JIT compiler, scheduler and predictive modelling approach. This is followed by our experimental setup and results. Related work and a summary concludes the chapter.

6.2 Motivation

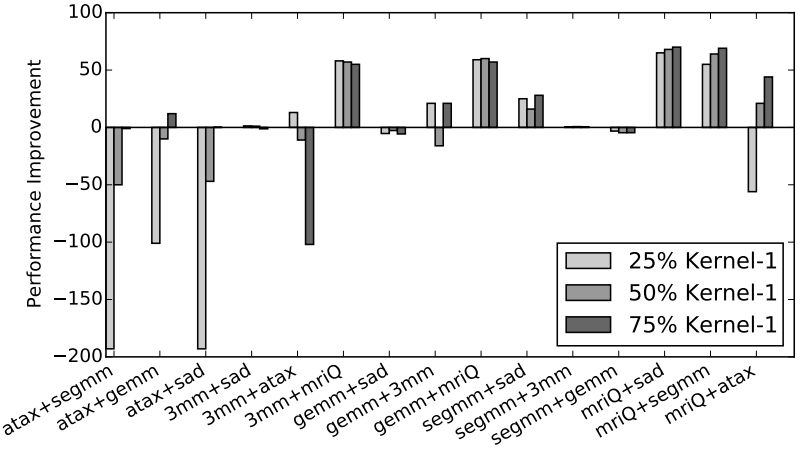
This section shows how sharing a GPU and effectively utilizing the CPU can improve performance for multi-program OpenCL workloads. We first examine six benchmarks, from the Polybench and Parboil benchmark suites and show that, individually, they have widely divergent performance on a GPU. We then show that concurrent execution can improve performance but existing approaches perform poorly. We then show that by incorporating concurrent GPU execution of kernels with combined CPU/GPU scheduling we can outperform existing approaches proposed in prior chapters.

Single kernel GPU performance Figure 6.1a shows the speedup of a single kernels on a NVIDIA GPU over CPU execution time. Some kernels, such `mriQ`, `gemm` and `3mm` `segmm`, have significant speedups on the GPU, however, for others e.g. `sad` and `atax`, they slowdown. Knowing which device to use improves application performance.

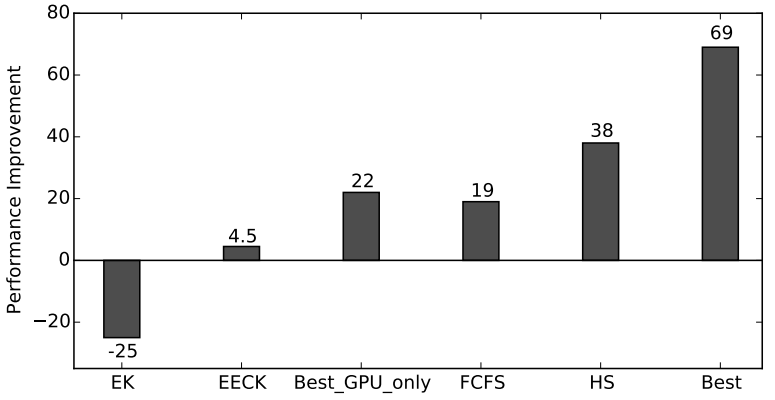
Concurrent Kernel Execution Concurrent execution of kernels on a GPU by merging them can improve performance. The improvement depends on the programs selected and how much of the GPUs resources are allocated to each as shown in fig-



(a) Single kernel speedup on GPU over on CPU.



(b) Performance improvement by concurrent kernel execution. (result in percentage)



(c) Performance improvement by different multi-task execution approaches. (result in percentage)

Figure 6.1: Multi-Kernel execution on CPU+GPU platform

ure 6.1b (the same example that is used in Chapter 5). Here the x-axis denotes the pair of programs merged and the y-axis, the performance improvement over running the kernels non-concurrently. Each pair has three performance bars. The first corresponds to 25% of the GPU allocated to the first program (the rest allocated to the second program). The second and third bar correspond to 50% and 75% respectively. The performance of many merged kernels is poor and only improves in certain cases. In figure 6.1b, no matter how `atax` and `segmm` are combined, their performance is always worse than running these two kernels sequentially. This is due to limited GPU resources. Kernel pairs, such as `3mm+mriQ`, `gemm+mriQ`, `segmm+sad`, `mriQ+sad` and `mriQ+segmm`, experience a higher throughput when running concurrently

Other kernel pairs such as `3mm+atax`, `gemm+3mm` and `mriQ+atax`, can achieve good performance but the correct allocation of resources is critical otherwise they will slowdown.

Existing Concurrent Approaches In figure 6.1c, the first three bars correspond to three approaches to concurrent execution. Elastic-Kernel [Pai et al. (2013)] (EK) merges pairwise without regard to suitability. Energy-Efficiency Concurrent Kernel [Jiao et al. (2015)] (EECK uses prior profiling to determine what to run concurrently, ahead of time. To make the use of profiling realistic, we use profiling information from a small data set to guide merging. Best_GPU_only represents the best performance available by choosing the right kernels to execute together and represents an upper-bound on performance. Figure 6.1c shows the speedup of each approach over just running the kernels sequentially on the GPU. EK suffers a 25% slowdown while there is a 4.5% improvement when using EECK.

For EK, the slowdown is caused by carelessly selected kernel pairs and suboptimal mixing ratio selection. EECK is sensitive to the accuracy of the profiling. More accurate profiling would certainly help but its excessive cost cannot be justified in a multi-tasking environment. The third bar in figure 6.1c shows that there is a potential 22% performance improvement available when merging kernels smartly.

Separate vs Concurrent Kernel Scheduler GPU based systems have host multi cores which are also scheduling targets as described in Chapter 4. The last 3 bars in figure 6.1c show the performance when using different scheduling policies that use the CPU over the performance available when just executing kernels sequentially on the GPU. FCFS is a first-come-first-served scheduler that gives 19% improvement and is

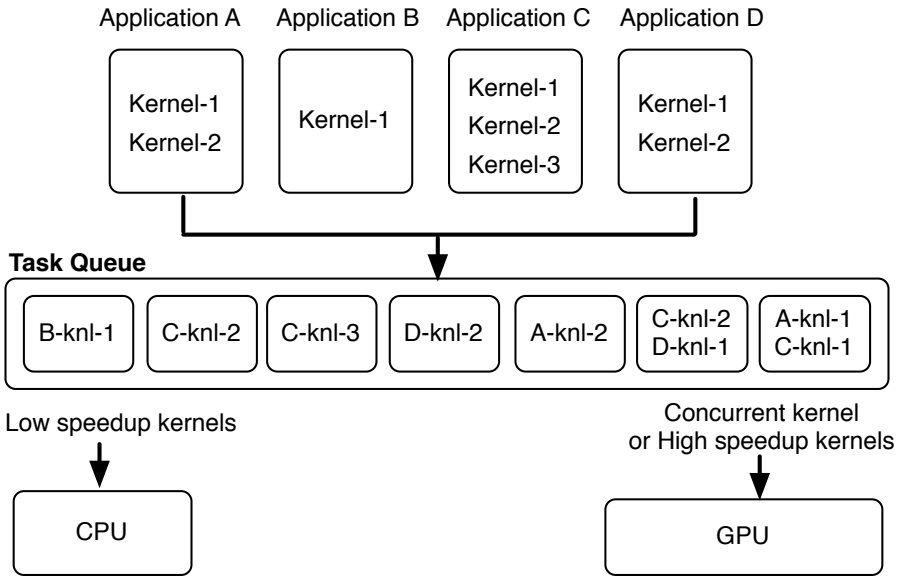


Figure 6.2: Multi-task scheduling on CPU+GPU platform.

less than the best performance of using just GPU alone. The Heterogeneous-Scheduler (HS) [Chapter 4] can achieve a significant improvement, 38%. If, however, we were able to correctly determine which kernels to merge and which kernels to schedule to the CPU we can achieve a 69% improvement as shown by last bar labelled Best.

In summary, both concurrent kernel execution and scheduling to CPU and GPU are able to improve system performance. Furthermore, there is significant room for improvement over existing schemes. In this chapter, we propose a runtime framework together with a Just-In-Time compilation tool to create and schedule concurrent kernels to CPU/GPU heterogeneous platforms.

6.3 Overall Scheme

Figure 6.2 shows our overall scheme. Users compile their applications, which are then submitted to the smart runtime. It examines each kernel of each application and determines if it is best to merge it with another kernel from another application or to execute it separately. It then determines whether to schedule it on the GPU or the CPU and places the (merged) kernels in an ordered task queue. Tasks are dispatched from one end of the queue to the CPU, the other end to the GPU. Determining whether to merge or not depends on the other dynamically available kernels and is based on a model learnt offline.

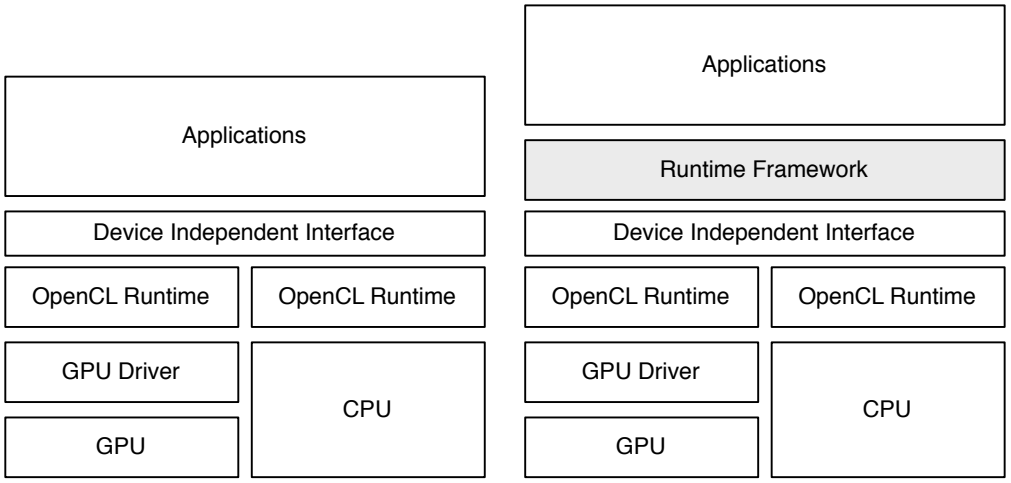


Figure 6.3: The runtime framework is a software layer that is built on top of OpenCL SDK. Instead of running directly on the device, applications register themselves to the framework, from where the workloads are scheduled to the device.

6.4 Runtime Framework

The runtime framework is a software layer sitting on top of the OpenCL runtime. Applications access target devices through this framework rather than directly. The core benefit of this scheme is that the software layer enables a set of optimizations which are transparent for the applications and their developers, such as eliminating collisions in accessing GPU device, balancing workload between CPU and GPU, and co-executing kernels on GPU to improve hardware resource utilization.

Figure 6.3 shows the runtime framework, as a software layer, built on top of the OpenCL runtime. Without this layer, every OpenCL application runs its host code on CPU and prefers to execute its kernel on the GPU. Since OpenCL is a low-level programming standard, the programmer has to select explicitly which target device the kernel code is going to run. The decision is usually fixed unless the developer puts more effort into making the program more adaptive. Hard coding programs to run on the GPU causes imbalances in processor usage and introduces collisions if more than one programs tries to execute on the GPU at the same time. However, instead of running on the device directly, if executing through this runtime framework, the applications can be remapped to any of the devices to improve hardware usage.

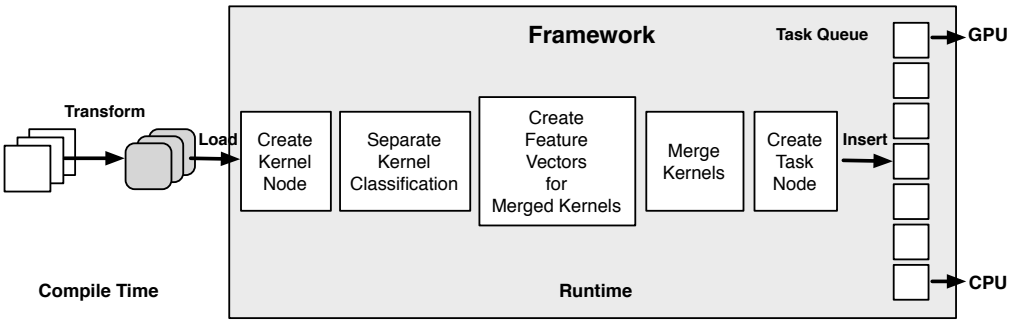


Figure 6.4: The main modules of the framework and their connections. Applications are transformed to dynamically loadable libraries. Then, they are loaded by the framework at runtime. The framework analyzes the information given by applications to create task nodes. Tasks nodes are inserted into the task queue, from which they are scheduled to the target devices.

6.4.1 Main Modules

Figure 6.4 presents the main modules of the framework. Applications are first transformed into dynamically loadable programs. Instead of running independently, applications pass kernels, with input data, and other information to the framework. After that, the framework evaluates the kernels to label their device affinity and creates the kernel pairs which have better performance when running concurrently on GPU. Finally, the kernels and kernel pairs are inserted into the task queue from where they are allocated to eitehr CPU or GPU through different ends.

There are five major modules in this framework: static feature extractor, kernel node creator, separate kernel classifier, feature vector constructor, kernel merging constructor, and task node creator. Figure 6.4 shows how these modules are connected to each other. Modules of static feature extractor and separate kernel classifier have been introduced in Chapter 4, and the kernel merging constructor has been presented in Chapter 5. Therefore, section 6.4.2 focuses on introducing the the other two modules which are kernel node and task node creators. Also, in section 6.4.2, we present a detailed description of this multi-threading framwork.

6.4.2 Multi-threading Design

OpenCL Work Flow

OpenCL provides a set of device-independent application programming interface (API) to offer a unified design flow for each program. The application is decoupled from the specific hardware, and instead of concentrating on the difference between the separate type of processors the programmer can focus on the APIs regardless of what devices the program is going to execute on. This feature of the OpenCL standard requires every program to have a similar workflow.

The first diagram in Figure 6.5 gives the details of the major API calls and their execution order. First, each program checks the platform and devices that the platform supports. Then, a context is created, which can have one or more devices. The command queue is used to communicate between the host and device. The host sends instructions via a command queue to execute a kernel on the device and transfer data back and forth. These API calls are mainly platform oriented.

Once the working environment is initialized, the application loads and compiles the kernel and prepares the input data in buffers. By writing the data to the device memory and setting the arguments for the kernel, the host then issues the compiled kernel to the device and read the results back once the kernel finished. The dynamically allocated memory and objects, such as kernel and kernel program, must be released to avoid memory leakage. Finally, before the application finishes, the platform environment has to be cleaned, e.g. release the command queue object. Here, most of the API calls are application related, as different applications have separate kernel implementations and input data.

Master-Worker Multi-threading Design

By analyzing the working flow, the framework is designed as a multi-threading runtime system. A master thread performs the main function of the framework. It creates a unified working environment for all arriving applications. Each OpenCL application that waits to be executed by a device is described as a worker thread that is loaded and managed by the master thread at runtime.

Figure 6.5 shows the API calls in master and worker thread in the middle and right diagram separately. The master thread, besides controlling the working environment setup and cleaning, loads applications as independent threads and manages their scheduling. For the application, instead of running directly, it prepares data and kernel

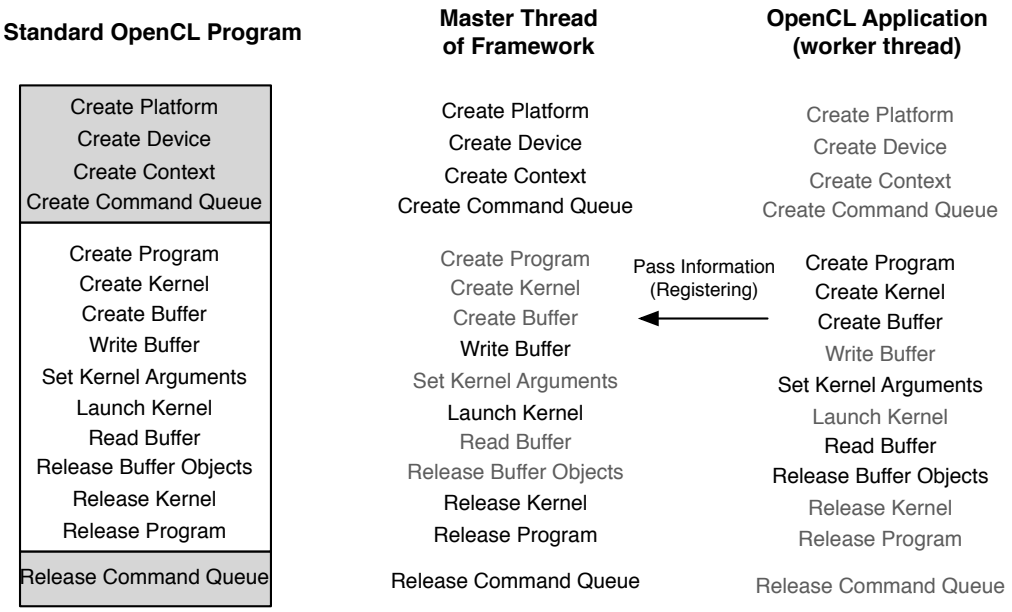


Figure 6.5: OpenCL Workflow

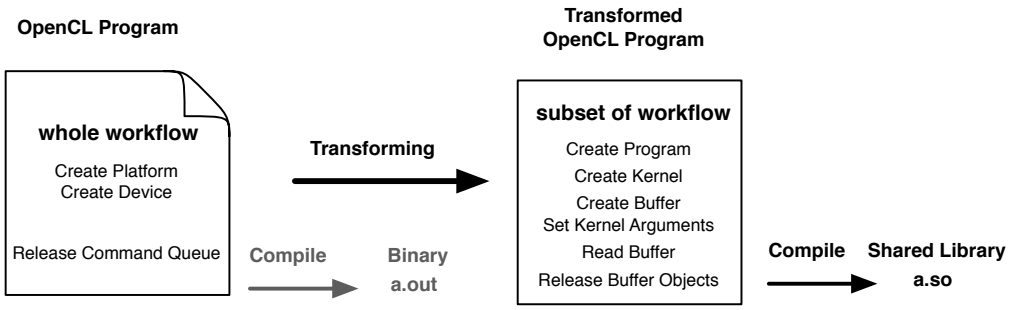


Figure 6.6: Dynamically Loadable Task

then registers them to the master thread, from which the data memory will be written, and the kernel will be executed.

The master-worker multi-threading design of the framework has two advantages. First, the unified platform setup eliminates the cost spent on hardware initialization for each application, especially when the time on initialization outweighs the time of kernel execution. Second, the multi-threading design provides a unified address space, which is significant, as the input data loaded by separate applications can be passed to the master thread via an address pointer and thereby avoids unnecessary data copying.

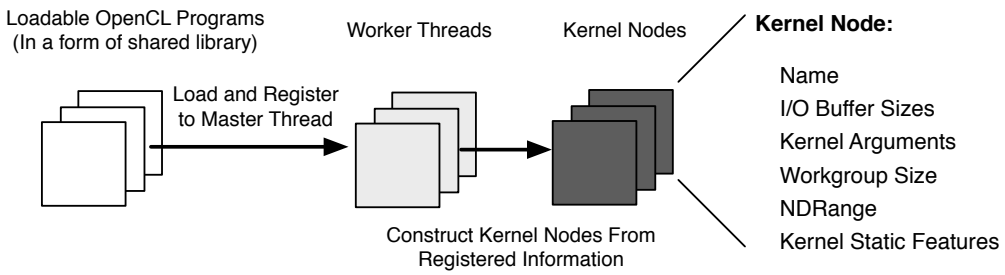


Figure 6.7: Kernel Node

Reloadable Application Constructing

We compile the OpenCL program as a shared library, instead of a binary, as part of the API functions have been removed from the source code. Figure 6.6 illustrates the transformation. As a shared library, the OpenCL program can be loaded by the master thread dynamically and shares the same address. It enables the master thread to create a kernel node for each application and classify the program with the help of machine learning based classifiers.

Kernel Node Construction

The worker thread prepares input data and compiles the kernel. It then registers its information with the master thread. The core information from the worker thread contains the size of input/output data, the name of the kernel, the arguments of the kernel, the workgroup size of the kernel, the NDRange of the kernel, and the kernel's code features. With all these details about the application, the framework master thread creates a kernel node of the application. The procedure is shown in figure 6.7.

Task Node Construction

The task node is created from the kernel node with extra information about a single kernel's estimated device affinity and co-execution kernel mixing ratio. The two kinds of task nodes: single and concurrent task node, are labeled separately by the machine learning-based classifiers presented in Chapter 4 and 5. Figure 6.8 shows the form of these task nodes. The task node is the basic unit that is inserted into the task queue. The location of the task node in the queue determines when and where the task is going to execute.

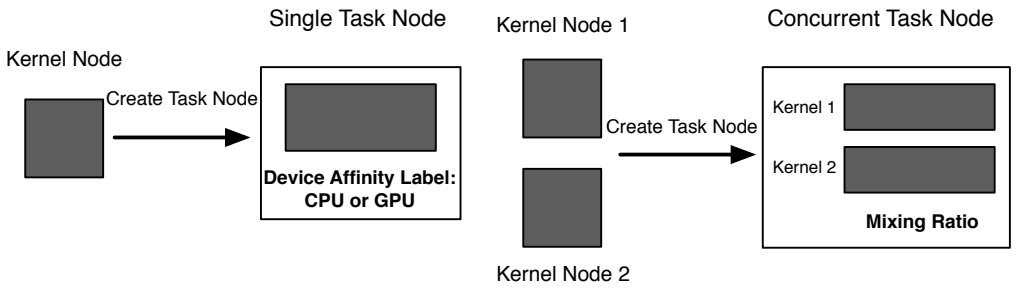


Figure 6.8: Single task node is built from single kernel node by adding the information of estimated device affinity. Concurrent task node is created from two kernel nodes together with their mixing ratio.

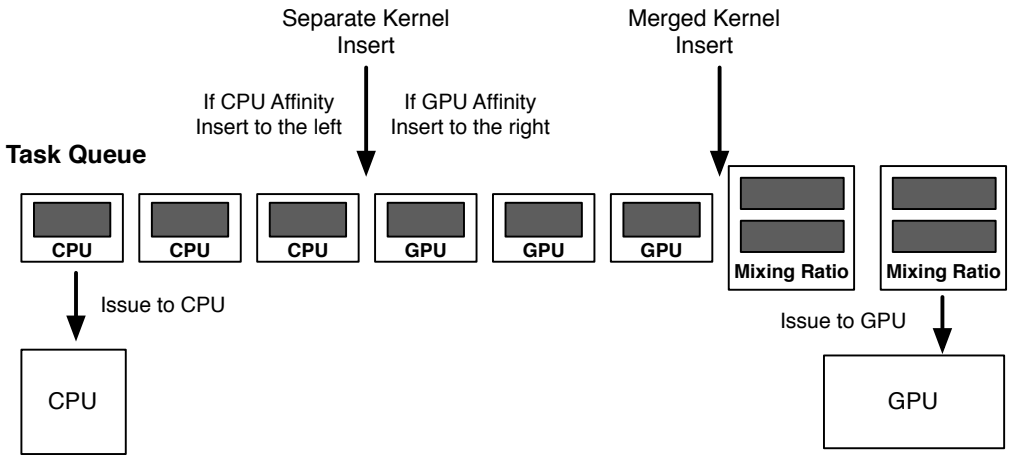


Figure 6.9: Task Queue

Inserting Task Node into Task Queue

The task queue is shown in figure 6.9. The single task node which has an estimated CPU affinity is queueing towards the left-hand side of the queue, which is connected to a multi-core CPU. The concurrent task node and single task node that has a GPU affinity are queued toward the other end of the queue, which is connected to the GPU processor.

New arriving tasks are inserted from some places that are in the middle of the queue. There are two insertion places, which are maintained and updated by the master thread. Figure 6.9 shows these two insertion places in arrows, assuming places in the queue are indexed in ascending order, and the lowest index is on the left most. Also, assuming the index of insertion place for the separate kernel is `n` and the other insertion place is `m`. When a new task is arriving, if it is predicted to have a CPU affinity, then

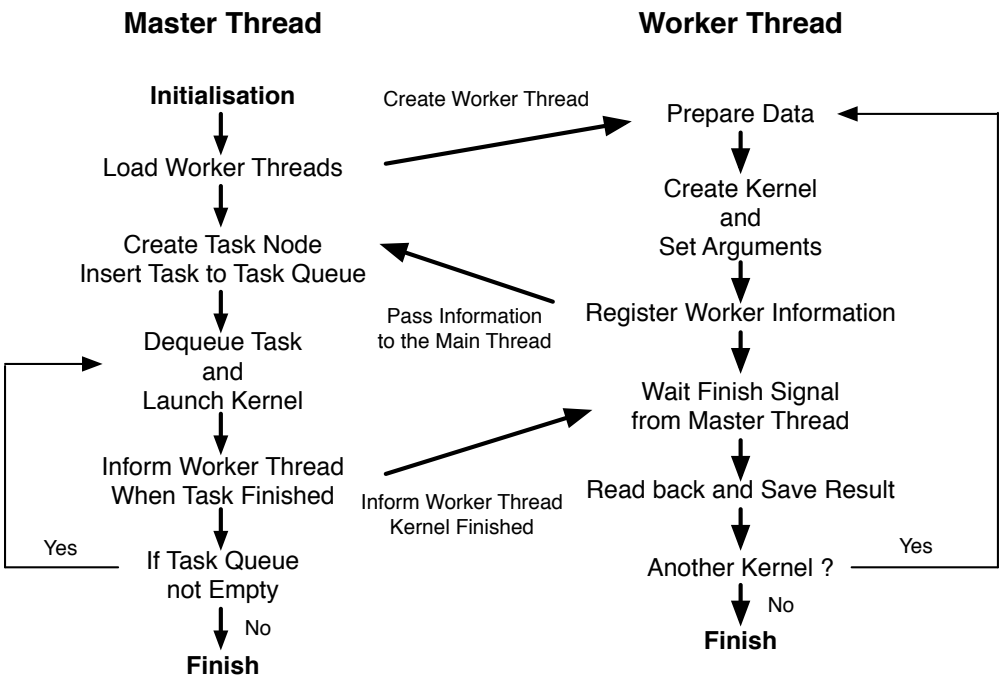


Figure 6.10: Synchronization

this task is inserted in the place of `n-1`. If this task is estimated to have a GPU affinity, the kernel merging classifier then examines all the other GPU affinity tasks to check whether or not the new arriving task can be paired up. If a pair is found, then a new concurrent task node is created and inserted at the place of `m+1`. Otherwise, this task is a single GPU affinity node which will be inserted at the place of `n+1`.

Kernel Synchronization

The master and worker threads must be synchronized to keep a valid performing order, particularly for the applications that have multiple kernels or have one kernel being iterated many times on the input data. Figure 6.10 shows how the thread synchronization works.

The master and worker threads are synchronized via signals. Before any applications are loaded, the master thread of the framework must be first started to initialize the system. Then the master thread loads the applications concurrently. The loaded applications then prepare their data, kernel and other information. Once the preparation is ready, the worker signals the master and registers its information with the master to create a kernel node for it. After that, the worker stalls itself until the master sends a signal to inform it that the kernel is completed. Then it checks whether or not there

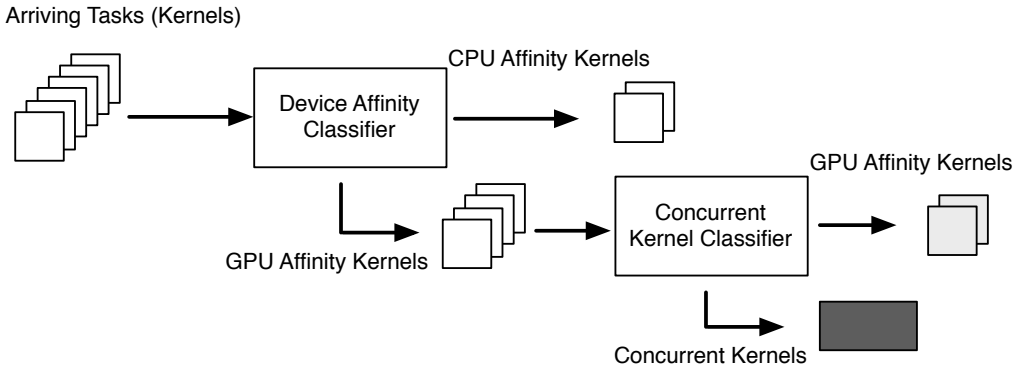


Figure 6.11: Model

is another kernel coming behind or the same kernel is iterated again. In either case, the worker signals the master to update the register information and stalls itself again until this time the kernel is completed. The master uses the register information from the workers to create task nodes and inserts them into the task queue. Once a job is finished, it informs the related worker and checks whether or not there any other jobs in the queue. If the queue is not empty, the master selects one and dequeues it to the newly idle device. If there is no job exists in the queue, then the framework terminates itself after a system cleaning up.

6.5 Model

In this chapter, we use both the machine learning based classifiers designed in chapter 4 and 5 to classify arriving tasks and insert them into the sorted task queue. Figure 6.11 shows how the two-step classifier works. For the arriving tasks, we first examine them by the device affinity classifier to detect whether they have CPU or GPU affinity. As explained earlier, the tasks with CPU affinity are inserted into the task queue right after the estimation. The rest of the tasks are passed to the concurrent kernel classifier, which estimates the co-running performance by exhaustively pairing up with all other kernels. Then, the maximum matching algorithm is performed on this graph to find out the maximum number of kernel pairs. For the kernels that have GPU affinity but do not have a better performance by co-running with any other kernel, the framework queues them in the task queue, towards the GPU end. After that, the merged kernels are inserted into the queue and these kernels will be launched to the GPU first before all single kernels.

6.6 Experiment Setup

We compare our approach to a number of approaches on two platforms. We use the same experiment setting as the one used in Chapter 5. For the convenience of reading, the detailed setting and benchmark are listed here again.

Elastic Kernels (EK) This approach runs two kernels concurrently and merges the corresponding host programs. It does not have a model to determine what to merge and does not use the CPU as a scheduling target. It co-executes all kernels in pairs.

Energy-Efficient Concurrent Kernels (EECK) This approach is similar to EK but requires profiling of the applications beforehand to determine what to merge. It does not use the CPU as a scheduling target. It uses profiling information from a small data set to guide kernel mergeings at larger data set.

Separate or Concurrent on GPU(SoC_GPU) Our approach to concurrent execution of kernels without using the CPU as a scheduling target

First-come-first-served(FCFS) This is a simple scheme that schedules jobs to either the CPU or GPU based on availability. It does not run kernels concurrently.

Heterogeneous Scheduling (HS) This uses a model to schedules jobs to either the CPU or GPU based on availability. It does not run kernels concurrently

Separate or Concurrent on GPU (SoC) Our approach to both concurrent execution of kernels and using the CPU as a scheduling target

To make a fair comparison, we implement EK and EECK and ignore the introduced overhead *i.e.* the cost of profiling and recompilation. Such overheads usually outweighed the benefits, making them hard to work in practice. Throughout the comparison, we use a unified metric, which is system throughput, to evaluate the results. The experiments are carried on a number of benchmarks from Parboil and Polybench benchmark suites.

6.6.1 Platform and Benchmarks

We evaluate on two CPU+GPU systems. Both have an Intel Core i7 4-core CPU and 16GB main memory. One platform contains an NVIDIA GeForce 780 and the other one contains an AMD HD 7970, see table 5. Both systems host OpenSUSE 12.3 Linux. We use LLVM 3.4 for JIT compilation and benchmarks are compiled using GCC 4.7.2 with -O3 option.

We restrict our attention to benchmarks with 1D and 2D NDranges from two main-

Table 6.1: Hardware platform

	Intel CPU	**NVIDIA GPU**	**AMD GPU**
Model	Core i7 4770K	GeForce GTX 780	Radeon HD7970
Architecture	Haswell-DT	Kepler GK110	Tahiti XT
Core Clock	3.4 GHz	1215 MHz	1000 MHz
Core Count	4 (8 w/ HT)	2304	2048
Memory	16 GB	3 GB	3GB
Memory Bandwidth	21GB	288 GB	264 GB

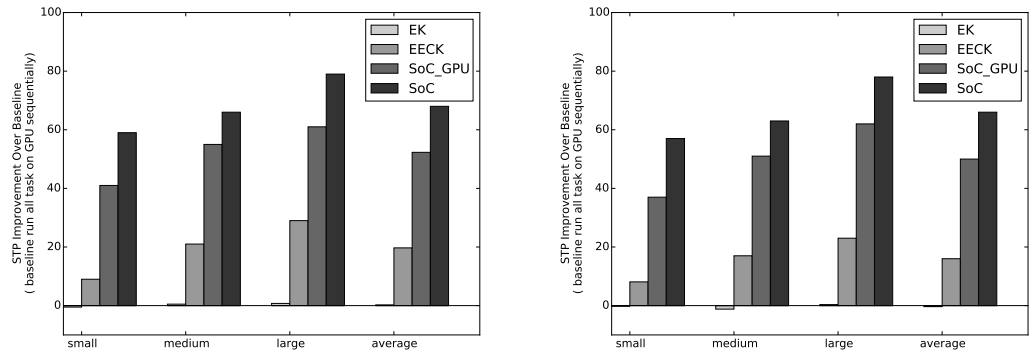
Table 6.2: Benchmarks

Suite	Benchmarks	Benchmark
Parboil	BFS	Mri-Q
	Sgemm	Spmv
	Sad	
Ploybench	ATAX	BICG
	CORRELATION	GESUMMV
	SYR2K	SYRK
	2DCONV	3DCONV
	GEMM	GRAMSCHMIDT
	2MM	3MM
	COVAR	FDTD-2D
	MVT	

stream OpenCL benchmark suites: the Parboil and the Polybench benchmark suite giving 20 programs in all. The benchmarks we used in this chapter are shown in list 6.2

6.6.2 Performance Evaluation

We evaluated our schemes with 500 different task configurations. We selected 10 different task queue sizes containing between 2 and 64 kernels. For each task queue size, we randomly selected 50 different programs, to give 500 configurations. As behaviour is dynamic, we evaluated each configuration 30 times and report the median performance. This results in 6000 experiments per policy. Performance is presented throughout as speedup relative to executing just on a GPU i.e. the STP metric.



(a) SoC vs. concurrent kernel execution on NVIDIA platform

(b) SoC vs. concurrent kernel execution on AMD platform

Figure 6.12: Summary of performance improvement

6.7 Results

In this section, we evaluate our approach against alternative approaches and analyze the behavior and accuracy of our predictive models. We then examine the impact of our runtime framework on performance.

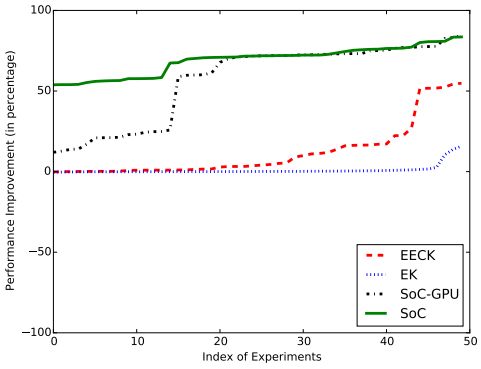
6.7.1 Performance Improvement Over Alternatives

In this section, we compare the result of our approach against other state-of-the-art methods on both NVIDIA and AMD platforms. The SoC and SoC_GPU use the runtime framework described in this chapter; The other alternatives are not running on this runtime layer. Instead, they work exactly in the way how they were proposed in their original papers.

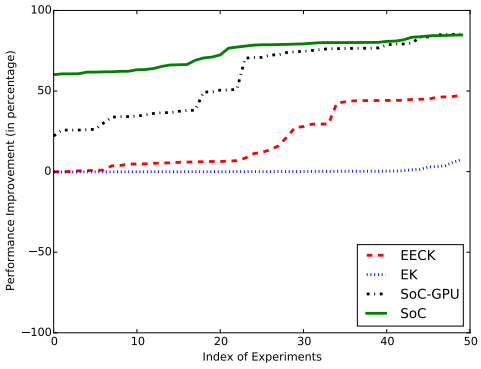
Comparison to Concurrent Kernel Execution

Figure 6.12 summarises the results of SoC/SoC_GPU compared to the concurrent kernel implementations. Except for EK, the performance of all the other methods improves as the number of tasks increases. SoC has the best throughput. On average, it is 16% better than SoC_GPU on the Nvidia platform and 16% on the AMD platform. Comparing to EECK, SoC is 49% and 50% better on Nvidia and AMD platform separately.

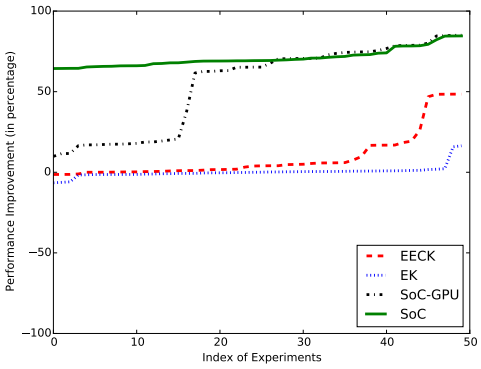
Figure 6.13 shows the results in greater detail of our approach against EECK, EK, and SoC_GPU, which use only the GPU to serve concurrent kernels. On each platform,



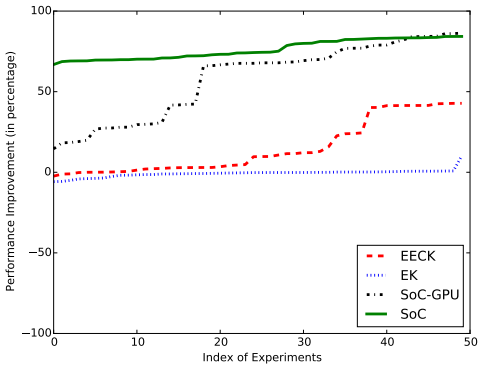
(a) 16 concurrent kernels on NVIDIA platform



(b) 32 concurrent kernels on Nvidia platform



(c) 16 concurrent kernels on AMD platform



(d) 32 concurrent kernels on AMD platform

Figure 6.13: Performance improvement over baseline

the results of two task size configurations are shown separately. In the first configuration, 16 kernels run on the platform and in the second configuration increases to 32. On both platforms, SoC and SoC_GPU have a much stronger performance improvement over the baseline than EECK and EK.

When the number of tasks is increased from 16 to 32, most of the four approaches improve, except EK, which instead suffers a slowdown. On the AMD platform, the performance improvement delivered by EECK over the baseline has been raised from 9% to 15% as the number of tasks increases. For SoC_GPU, the corresponding performance change is 53% and 58%. By adding a CPU as a scheduling target, SoC provides the best performance, which is 70% and 76% better than the baseline when there are 16 and 32 tasks running. Randomly pairing up kernels for EK gives a slowdown from 0.23% to -0.7%, though more having tasks provide more options in kernels pairing up.

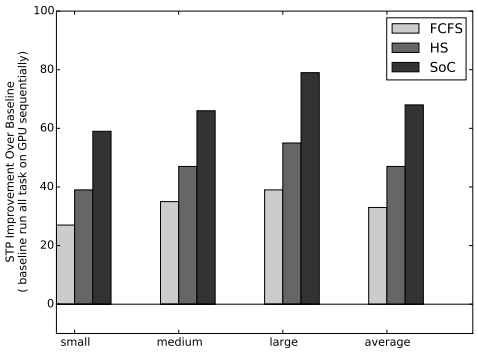
The results on the NVIDIA platform shows similar trends. For EK, performance improvement decreases from 1% to 0.5% when the number of tasks increases from 16 to 32. The other approaches all have a performance improvement when the number of tasks raises. The average performance improvement of EECK in 16 and 32 are 12% and 20%. The SoC_GPU has better performance, which is 56% and 58%. Finally, the best performance is again achieved by SoC, which is 69% and 73%.

When there is a small number of kernels, SoC_GPU may provide competitive performance comparing to SoC as long as there are enough kernel pairwise options; otherwise, performance is much worse than SoC. With the number of tasks increasing, SoC outperforms SoC_GPU continually, as the contribution of a CPU grows. For SoC_GPU, the worst performance of a large number tasks is better than with a small number of jobs. As can be seen in figure 6.13, for both NVIDIA and AMD platform, the curve gap between SoC and SoC_GPU grows wider from 16 tasks to 32 tasks, and the worst performance for 32 jobs is better than 16 jobs.

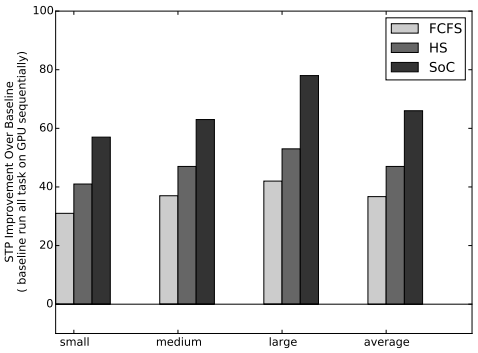
Comparing to CPU-GPU Scheduling

Figure 6.14 summarises the results of SoC compare to the separate kernel scheduling. All the methods improve as the number of tasks increases. SoC has the best throughput. On average, it is 21% better than HS on the Nvidia platform and 19% on the AMD platform. Comparing to FCFS, SoC is 35% and 33% better on Nvidia and AMD platform separately.

Figure 6.15 shows the results of SoC compared: HS and FCFS, which also use the CPU. On both NVIDIA and AMD platform, the performance improvement over

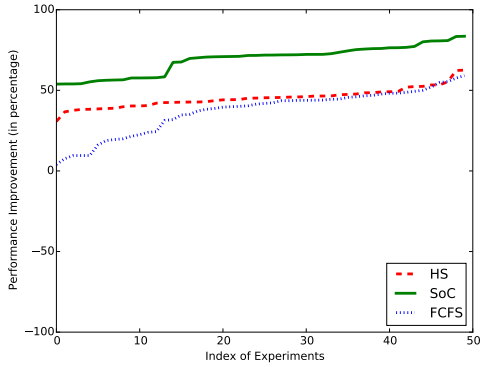


(a) SoC vs. separate kernel scheduling on NVIDIA platform

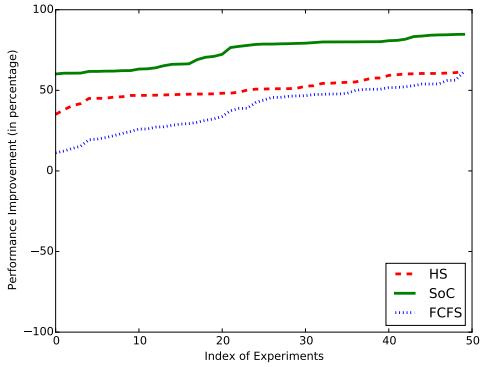


(b) SoC vs. separate kernel scheduling on AMD platform

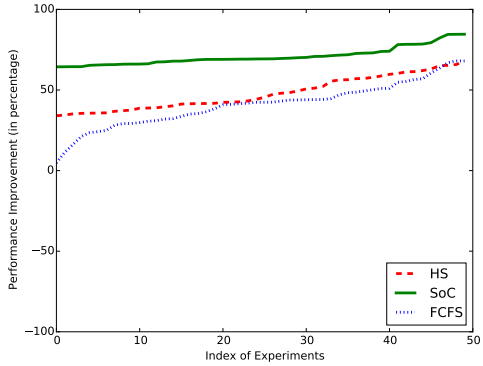
Figure 6.14: Summary of performance improvement



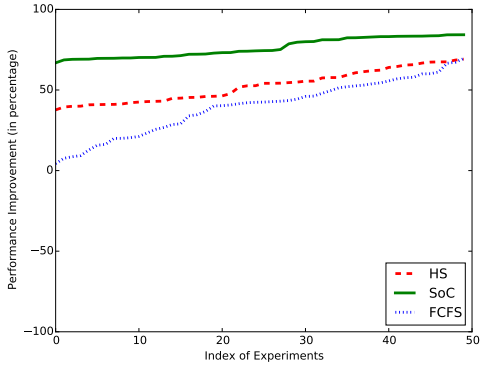
(a) 16 concurrent kernels on NVIDIA platform



(b) 32 concurrent kernels on Nvidia platform



(c) 16 concurrent kernels on AMD platform



(d) 32 concurrent kernels on AMD platform

Figure 6.15: Performance improvement over baseline

baseline is stable for both SoC and HS. However, the results of FCFS have spread in a wide range. As FCFS randomly schedules a task on CPU and GPU whenever it detects a processor becomes idle, the outcome of the scheduling is random as well. Hence, for 16 tasks scheduling, the worst performance improved by FCFS is 1% on NVIDIA platform and 1.7% on AMD platform, but the best are 51% and 53%. When the number of tasks is increased to 32, the result of FCFS scheduling does not change much.

On average, the performance improvement over baseline is 40% by FCFS, 47% by HS, and 70% by SoC for 16 tasks scheduling on the AMD platform. The corresponding result on NVIDIA platform is 36% by FCFS, 45% by HS, and 69% by SoC. When it come to 32 tasks scheduling, the results are 39% (FCFS), 52% (HS), and 76% (SoC) on the AMD; 38% (FCFS), 51% (HS), and 77% (SoC) on NVIDIA.

6.7.2 Performance Improvement Over Alternatives on The Same Framework

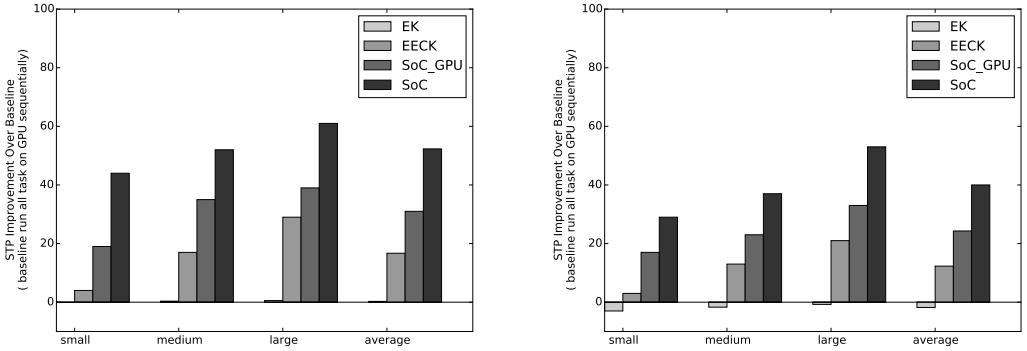
As our framework provides a unified control on hardware resources management, it improves the performance of our method. In order to isolate the contribution of SoC scheduling proposed in this chapter, a deeper comparison is carried out in this section by running all alternative approaches, including the baseline, with our runtime framework.

Comparing to Concurrent Kernel Execution

Figure 6.16 summaries the results of SoC/SoC_GPU compared to the concurrent kernel implementations. Except EK, the performance of all the other methods improves with the number of tasks increasing. SoC has the best throughput. On average, it is 11% better than SoC_GPU on the Nvidia platform and 16% on the AMD platform. Comparing to EECK, SoC is 36% and 28% better on Nvidia and AMD platform separately.

Figure 6.17d shows the results of SoC against other concurrent kernels when using the same framework. Though the cost on initialization has been improved by the framework for EK and EECK, SoC and SoC_GPU are consistently providing better performance comparing to them.

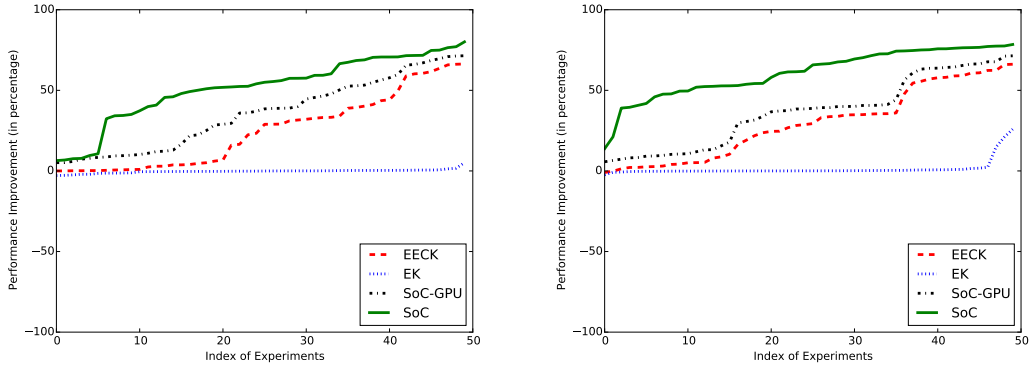
For a small number of tasks, 16 kernels, the average performance improved by EK is -0.2%, which is worse than the baseline. For EECK and SoC_GPU, the average improvements are 24% and 35%. SoC provides the best average result, which is 51%



(a) SoC vs. concurrent kernel execution on NVIDIA platform

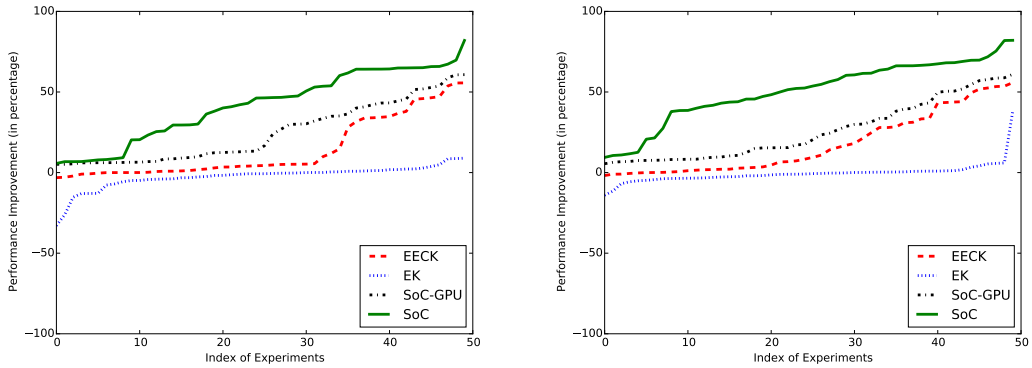
(b) SoC vs. concurrent kernel execution on AMD platform

Figure 6.16: Summary of performance improvement



(a) 16 concurrent kernels on NVIDIA platform

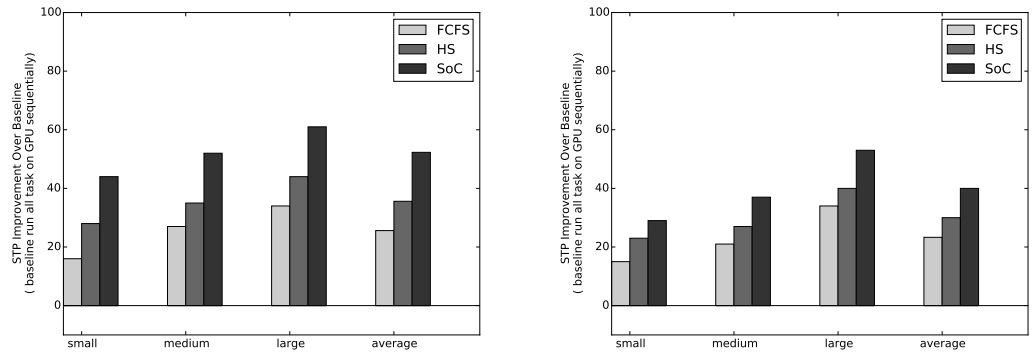
(b) 32 concurrent kernels on Nvidia platform



(c) 16 concurrent kernels on AMD platform

(d) 32 concurrent kernels on AMD platform

Figure 6.17: Performance improvement over baseline



(a) SoC vs. separate kernel scheduling on NVIDIA platform

(b) SoC vs. separate kernel scheduling on AMD platform

Figure 6.18: Summary of performance improvement

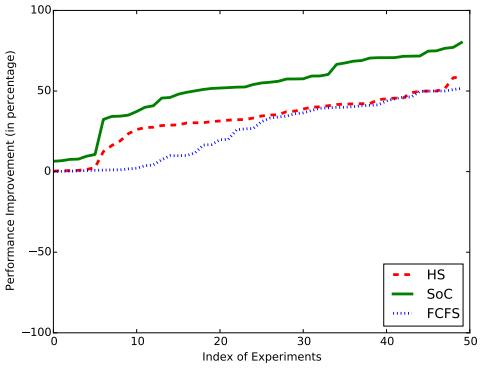
better than the baseline. For a larger number of tasks scheduling, e.g. 32 jobs, EK provides a result of 1.37% and result from EECK is 29%. For SoC_GPU and SoC, their results are 36% and 60%. All above results are collected from the NVIDIA platform.

On AMD platform, for 16 task scheduling, the results for EK, EECK, SoC_GPU, and SoC are -2.6%, 14%, 24%, and 41%. For 32 tasks scheduling, the corresponding results are -0.5%, 17%, 25%, and 51%.

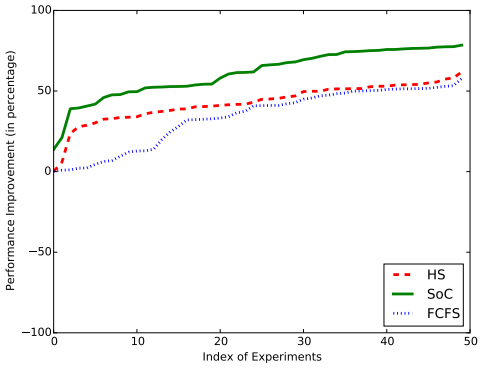
Comparing to CPU-GPU Scheduling

Figure 6.18 summarises the results of SoC compare to the separate kernel scheduling. All the methods improve as the number of tasks increases. SoC has the best throughput. On average, it is 17% better than HS on the Nvidia platform and 10% on the AMD platform. Comparing to FCFS, SoC is 27% and 17% better on Nvidia and AMD platform separately.

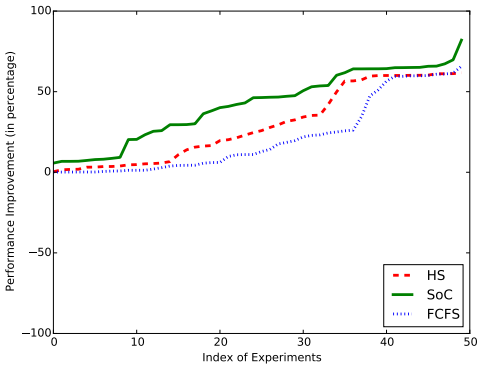
Figure 6.19 gives the results of SoC comparing to the FCFS and HS. On average, the performance gap between SoC and HS becomes widens as the number of tasks increases. Similar trends exist between SoC and FCFS as well. When there are 16 tasks, the average performance improvement over baseline is 25% (FCFS), 32% (HS), and 51% (SoC) on the NVIDIA platform. The corresponding results on AMD platform are 21%, 29%, and 41%. The average performance all increases as the number of task rises. For 32 tasks scheduling, the performance improvements are 33% (FCFS), 42% (HS), and 60% (SoC) on the NVIDIA platform, and 30% (FCFS), 36% (HS), and 51% (SoC) on the AMD platform.



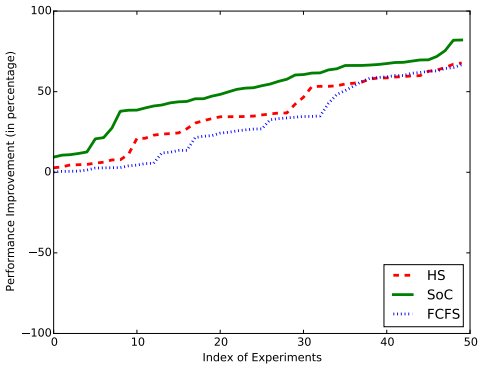
(a) 16 concurrent kernels on NVIDIA platform



(b) 32 concurrent kernels on Nvidia platform



(c) 16 concurrent kernels on AMD platform



(d) 32 concurrent kernels on AMD platform

Figure 6.19: Performance improvement over baseline

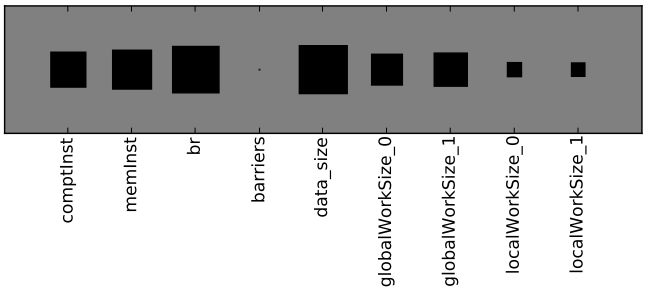


Figure 6.20: Feature importance on separate kernel speedup classifier. The bigger the box is, the more important the feature is.

6.7.3 Performance Improvement by the Framework

Without the framework, SoC scheduling outperforms the baseline by 68% on the Nvidia platform, and 66% on the AMD platform. When the baseline also uses the same framework, the average performance improvement of SoC become 52% and 40%. Therefore, the framework developed in this chapter provides 16% performance improvement on both Nvidia and AMD platforms.

6.8 Analysis

6.8.1 Estimation Accuracy

In our system, we trained our classifier using a leave-one-out-cross-validation, on 38 distinct kernels and 2031 concurrent kernels with different mix ratios. For the CPU/GPU classifier, we have an accuracy of 88% on NVIDIA and 90.3% on AMD platform. For the concurrent kernel classifier, its accuracy is 81% and 85% on those platforms.

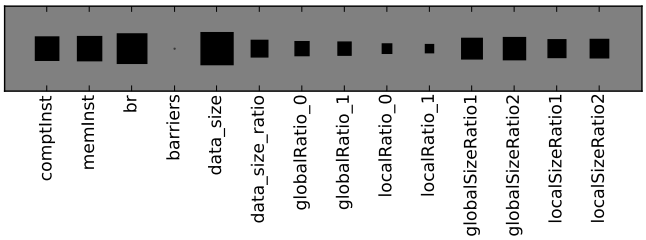


Figure 6.21: Feature importance on concurrent kernel classifier. The bigger the box is, the more important the feature is.

6.8.1.1 Feature Importance

We have evaluated feature importance for both of the classifiers. Features play an important role in the prediction accuracy, as they represent the characteristics of kernels. Various features impact the predictor accuracy in different ways. Figure 6.20 and 6.21 show the detailed information about our features importance in each of the predictors.

The feature extractor is upgraded in this chapter. Therefore, features shown in Figure 6.20 are different from those in Figure 4.6. The main difference is that we separate branch instructions from the computation instructions, as they impact kernel performance in different ways. On the contrary, the static feature of math functions which is shown in Figure 4.6 is not an independent feature anymore. Such kind of math functions feature has been weighed (according to Table 5.2) and added to the feature of computation instructions. Finally, since the feature of the number of blocks in Figure 4.6 can be calculated from the number of global and local thread, we can safely remove it from the feature vector. By upgrading the feature vector, our predictor has an enhanced accuracy than the one that proposed in Chapter 4.

In general, there are some critical features, such as the data size, branch instruction ratios, memory access instruction ratios and computation instruction ratios. They are important because they directly link to the kernel performance. For example, a kernel that has high computation instruction ratio on a reasonable size of data and with fewer

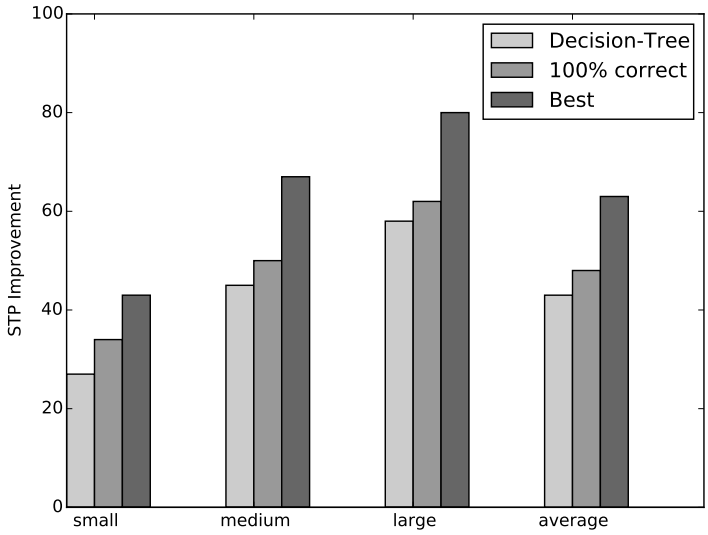


Figure 6.22: Performance limit study on NVIDIA platform. On average, a 100% accurate classifier can improve the performance by another 5%. If we have more information about the task, such as kernel execution time, the performance could be improved by 20%.

branches usually achieves high performance when it runs on GPU.

Other features impact the accuracy in a different way. For single kernel speedup classifier, the global ND-Range has more influence than the local ND-Range, however, for a concurrent kernel classifier, this influence gap is not as big as for separate kernels. The kernels ND-Range mixing ratios have an effect on concurrent kernels classification, as they can help filter out some of the extreme mixing cases. The feature of barrier contributes the least in our classifier because the barrier is rarely used in the benchmark kernels.

6.8.2 Limit Study

To examine how the classifier accuracy impacts the performance, we performed a limit study. We evaluated the system throughput in three different cases. In the first case, we use our learnt models. In the second case, we replace the models with predictors with 100% accuracy. They always correctly determine whether a kernel runs faster on a GPU or a CPU and whether or not it runs faster concurrently or not.

In the third case, for each sequence of tasks, we run 10,000 different scheduling orders to test the potential performance upper bound. Because of combinatorial com-

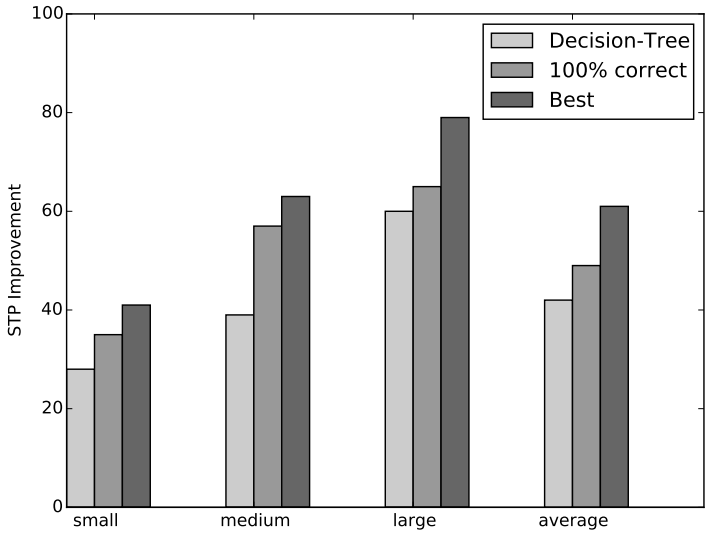


Figure 6.23: Performance limit study on AMD platform. On average, a 100% accurate classifier can improve the performance by another 7%. If we have more information about the task, such as kernel execution time, the performance could be improved by 19%.

plexity, we cannot evaluate all possible kernel combinations and scheduling orders, and this approach gives only an approximation of performance achievable. It represents the Best schedule found.

Figure 6.23 and 6.22 show our results on two different platforms. In general, the more accuracy a classifier is the higher throughput we can expect. For 100% accurate classifier, we can get a 5% performance improvement on NVIDIA platform and 7% on AMD platform over our classifier on average.

Classifying accurately is not enough. There may be a case where 2 kernels have large concurrent speedup but have insignificant execution time while 2 others have a small concurrent improvement but are long running and dominate overall execution time. This is shown by the performance of the best scheduler which improves STP further. On the NVIDIA platform, its performance is 20% better than our classifier and on AMD platform this performance improvement is 19%.

This shows that while our approach has significant improvement over existing schemes, there is still further room for improvement

6.9 Conclusion

In this chapter, we have proposed a runtime system and a JIT compiler to schedule multiple OpenCL kernels on a CPU-GPU heterogeneous platform. We have trained two predictive models from OpenCL kernel's static and runtime features to determine whether the kernels would be merged and launched together, or dispatched to the best-fit device separately. To evaluate the performance, we compare our approach with a wide range of state-of-the-art methods on two different heterogeneous platforms. On average, if only use the GPU device, our approach could improve system throughput by 27% and 20% on NVIDIA and AMD platform respectively. This improvement is more than 20% better than Elastic-Kernel and 11% better than Energy-Efficient-Concurrent-Kernel. When adding CPU as scheduling target as well, our approach improves the system throughput by 43% on NVIDIA and 42% on AMD platform. And this is about 15% better than the Heterogeneous-Scheduler. Though our approach achieves a significant improvement over existing methods, there is still further room for improvement. Precisely predicting task execution time would improve the performance further.

Chapter 7

Conclusion

This thesis has proposed several methods for addressing challenges of multi-task scheduling on CPU-GPU heterogeneous systems. It proposes a machine learning-based approach to determining OpenCL kernel scheduling. Chapter 4 introduces an individual kernel scheduler that dynamically allocates kernels to either multi-core CPU or GPU according to their predicted device affinity. Chapter 5 focuses on improving GPU hardware resource utilization smart space sharing. Chapter 6 introduces an efficient runtime framework that is able to combine individual and merged kernels scheduling. The contributions presented in this thesis have been implemented in software, which is portable across platforms with CPUs and GPUs from different vendors.

The structure of this chapter is organized as follows. Section 7.1 summarises of the contribution. Section 7.2 presents a critical analysis of the thesis together with the limitations of our method. Section 7.3 introduces the future work, and finally, we summarize this chapter in Section 7.4.

7.1 Contributions

This section summarizes the contributions of the three technical chapters.

7.1.1 Scheduling Kernels to the Best-fit Devices

A machine learning based classifier is developed in Chapter 4 to estimate the newly arriving OpenCL kernels device affinity. Device affinity is measured by a kernel's relative speedup, which represents to what extent running the kernel on GPU is faster than on multi-core CPU. The model is trained off-line on a broad range of training

samples that contain the kernel code features, runtime parameters, and pre-profiled relative speedups. At the runtime, the scheduler sorts the newly arriving kernels by estimating their relative speedup with the help of the pre-trained model. Kernels with high and low predicted speedup are queuing toward separate ends of the tasks queue, from which they will be issued to the connected devices. The object of the model is to learn the relationship between a kernel's characteristics and the target device and then use this knowledge to estimate affinities for unseen tasks. The scheduler manages all the OpenCL supported devices and optimizes the overall throughput via allocating task to the best-estimated device as well as overlapping and balancing the execution on CPU and GPU. This lead to an overall speedup of 20% over a random scheduling.

7.1.2 Co-running Kernels with the Most Appropriate Peers

A machine learning based kernel merging method is proposed in Chapter 5. Though concurrent executing kernels can improve GPU utilization, in practice, it does not necessarily mean an improved performance. Using simple metrics (such as computing intensity or number of branches) to guide co-execution of kernels do not bring the expected performance because of the complexity of kernels and target platforms. This chapter instead develops a machine learning based technique to detect kernels associativities. By learning from the kernel code features and the runtime parameters, the model estimates two kernels improved performance depending on mixing ratios. Some kernels work well together with many others. We use a graph matching algorithm in the scheduler to find the maximum number of kernel pairs. The proposed approach outperforms the state-of-the-art approach by 22%+ on both Nvidia and AMD platforms.

7.1.3 Runtime Framework for Mix Scheduling

The final contribution of this thesis is the development of a runtime layer that supports the efficiently mixed scheduling of separate and merged kernels on CPU-GPU heterogeneous systems. The framework manages all devices on the platform. It transfers data between host and device and issues kernels on behalf of the OpenCL applications registered to it. By estimating kernel device affinity and whether it can be effectively co-scheduled on the GPU, the framework merges the kernels and sorts them accordingly. As the framework provides a global environment for all OpenCL applications, it reduces the runtime overhead of setting up and initialization which normally has

to be done by every application. This scheduler outperforms the state-of-the-art concurrent kernels techniques by 50% and improves the throughput by 35% over random scheduling.

7.2 Critical Analysis

This thesis has presented contributions to improving the throughput of CPU-GPU heterogeneous systems; however, there are aspects of the approach that should be critically reviewed.

7.2.1 Unified Task Arriving Orders and Priority

In this thesis, all tasks are presumed to arrive at the same time and with the identical priority. It is a strong assumption, as the arrival order and priority have a significant impact on the scheduling decisions which affect the performance of both throughput and average turnaround time. Also, the scheduler in this thesis assumes each task occurs only once; however, in practice, some applications are loaded and executed periodically. Task arrival order and priority may not affect their device affinities, but require the scheduler to issue them in a different way. For example, when a kernel with high priority and GPU affinity arrives but neither CPU nor GPU is available, which task shall the scheduler preempt so as to execute this high priority job? To estimate and determine a suitable policy is a complex problem. We simplify the problem by assuming all tasks arrive simultaneously with the same priority, in the future, both of them needed to be taken into consideration.

7.2.2 Coarse Category of Classification

This thesis uses binary classifiers to estimate kernels device affinity and their associativity with other kernels. Speed of classification efficiency and accuracy are the main advantages of the binary classification. However, it cannot exploit all the potential performance that the platform promised. Section 4.10 shows that a finer classification can improve the performance further by a significant way. Relying on classification alone is a limited method, as kernel execution time cannot be estimated by a simple classification. Take separate kernel scheduling for example. A kernel will be labeled as CPU first task when its estimated relative speedup is 1.5x. However, this kernel may have the longest execution time among all its counterparts and its execution time

may contribute 90% of all tasks execution. Thus, instead of queueing it to CPU, the best strategy is to make it map to the GPU to reduce processing time. Using a coarsely grained classification, the scheduler can do nothing about it. Accurately predict kernel's execution time is hard and is our future work.

7.2.3 No Dependencies Between Kernels

This thesis assumes that all OpenCL kernels are independent, and from multiple users, so there are no dependencies between any two kernels. However, for some applications that contain multiple kernels, there is a high chance that one kernel's input is another kernel's output. When dependencies exist between kernels, merging and running them concurrently on GPU can optimize the hardware utilization as well as data movement. Some prior work has been done in this area. This thesis did not consider this situation, but prior work could be integrated into our framework, with an effort in extension.

7.2.4 Unaware of Coalesced Memory Access

Memory coalesced accessing has a significant impact on OpenCL kernels performance. However, how to summarize and weight coalesced or un-coalesced accessing with other static features is difficult. This is our future work as well.

7.2.5 Homogeneous Multi-core CPU processor

Finally, in this thesis, the target CPUs are homogeneous multi-core processors. However, there is a clear trend for the modern CPU processors to become heterogeneous as well. They contain cores with different hardware resources (such as ARM bigLittle) or integrated processors with a separate architecture, such as AMD Fusion and Intel i7 that contains integrated GPU. Increasing heterogeneity of the platform requires a more complex model, and therefore, the model could be the bottleneck of the system. Optimizing the model and abstracting the system effectively is a new challenge not addressed in this thesis.

7.3 Future Work

This section briefly introduces the directions for extending and improving the work presented in this thesis in the future.

7.3.1 Periodic Task and Priorities

As discussed earlier, this thesis assumes all tasks arriving simultaneously; however, in practice, the tasks require the device in different orders and have separate priorities and expected finishing time. Therefore, future work should consider scheduling orders and priorities. There are jobs that are periodically executed. Unlike a one-time job, periodic execution carries more detailed accumulated information about the task itself which can make the model more accurate. Hence, a more accurate model could be used in scheduling. It is a fruitful area of future work.

7.3.2 Workload Migration

Once the arrival order and priorities are considered by the scheduler, the jobs may need to migrate from one processor to another or be suspended, as a high priority may preempt a running task which has a lower priority. GPUs do not inherently support preemption; however, some work has been proposed to enable preemptive execution and interruption by extending the GPU hardware or software stack. As workload migration between CPUs and GPUs may introduce a large runtime overhead, a careful decision has to be made by the scheduler at runtime.

7.3.3 Finer Classification and Execution Time Regression

As pointed out in Section 4.10, overall throughput can expect a further improvement by using a model with a finer classification. However, without the information about the precise execution time, there is still a huge performance gap between the reality and the best that is achievable, even though a finer classification model may have been used. Therefore, to probe the impacts on performance, both finer classification model and execution time regression model should be examined.

7.3.4 Dependency Management

Dependencies among kernels affect their scheduling orders. However, merging kernels that have dependencies may enhance the performance by reducing unnecessary data transfers. The problem is, when and which kernels should be merged. This is future work.

7.3.5 Coalesced Memory Access Identification

As mentioned earlier, coalesced memory accessing has a significant impact on a kernel's performance. Currently, the training features have not included this information. Assuming all kernels are well written by the programmer and have optimized coalesced memory access is a strong condition. Therefore, to make the model more general, the pattern of memory accessing needs to be included in the feature vectors.

7.3.6 Heterogeneous Multi-core CPU

A multi-core CPU can be heterogeneous itself. Cores within the same CPU package can either share the same ISA but with different hardware resources (like cache sizes, depth of pipelines, or clock frequencies) or have different types of architectures (like integrated GPU). The heterogeneity of CPU brings more flexibility for the system, and at the meantime it also introduces extra complexity in constructing the model.

7.3.7 Power Aware Scheduling

So far in this thesis, the kernel merging and scheduling decision is made purely for the sake of performance improvement. In many areas, such as mobile system and data centre, power consumption is as equally important as throughput. In many cases, the goals of performance improving and power optimization conflict with each other, as typically running a program faster requires more power. Therefore, another aspect of the future work is to examine how to explore power consumption and performance efficiency.

7.4 Summary

This chapter presents the conclusion of this thesis. It first summarizes the contribution of each technical chapter. Then, it critically analyzed the limitation of the method proposed in this thesis. Finally, we introduced potential areas of future work.

Bibliography

Adriaens, J., Compton, K., Kim, N. S., and Schulte, M. J. (2012). The case for GPGPU spatial multitasking. In *18th IEEE International Symposium on High Performance Computer Architecture, HPCA 2012, New Orleans, LA, USA, 25-29 February, 2012*, pages 79–90.

Aguilera, P., Lee, J., Farahani, A. F., Morrow, K., Schulte, M. J., and Kim, N. S. (2014a). Process variation-aware workload partitioning algorithms for GPUs supporting spatial-multitasking. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014*, pages 1–6.

Aguilera, P., Morrow, K., and Kim, N. S. (2014b). Fair share: Allocation of GPU resources for both performance and fairness. In *32nd IEEE International Conference on Computer Design, ICCD 2014, Seoul, South Korea, October 19-22, 2014*, pages 440–447.

Aguilera, P., Morrow, K., and Kim, N. S. (2014c). QoS-aware dynamic resource allocation for spatial-multitasking GPUs. In *19th Asia and South Pacific Design Automation Conference, ASP-DAC 2014, Singapore, January 20-23, 2014*, pages 726–731.

Ahmad, W. and Narayanan, A. (2015). Artificial immune system: An effective way to reduce model overfitting. In *Computational Collective Intelligence - 7th International Conference, ICCCI 2015, Madrid, Spain, September 21-23, 2015. Proceedings, Part I*, pages 316–327.

Ahn, S., Park, S., Chertkov, M., and Shin, J. (2015). Minimum weight perfect matching via blossom belief propagation. *CoRR*, abs/1509.06849.

ARM (2016). ARM big.LITTLE technology. http://www.arm.com/products/processors/technologies/biglittleprocessing.php.

Augonnet, C., Clet-Ortega, J., Thibault, S., and Namyst, R. (2010). Data-aware task scheduling on multi-accelerator based platforms. In *16th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2010, Shanghai, China, December 8-10, 2010*, pages 291–298.

Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P. (2011). StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198.

Awatramani, M., Zambreno, J., and Rover, D. T. (2013). Increasing GPU throughput using kernel interleaved thread block scheduling. In *2013 IEEE 31st International Conference on Computer Design, ICCD 2013, Asheville, NC, USA, October 6-9, 2013*, pages 503–506.

Baghsorkhi, S. S., Delahaye, M., Patel, S. J., Gropp, W. D., and Hwu, W. W. (2010). An adaptive performance modeling tool for GPU architectures. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2010, Bangalore, India, January 9-14, 2010*, pages 105–114.

Baghsorkhi, S. S., Gelado, I., Delahaye, M., and Hwu, W.-m. W. (2012). Efficient performance evaluation of memory hierarchy for highly multithreaded graphics processors. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 23–34, New York, NY, USA. ACM.

Bakhoda, A., Yuan, G. L., Fung, W. W. L., Wong, H., and Aamodt, T. M. (2009). Analyzing CUDA workloads using a detailed GPU simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163–174.

Barak, A., Ben-Nun, T., Levy, E., and Shiloh, A. (2010). A package for OpenCL based heterogeneous computing on clusters with many GPU devices. In *Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS), 2010 IEEE International Conference on*, pages 1–7.

Becchi, M. and Crowley, P. (2006). Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Proceedings of the Third Conference on Computing Frontiers, 2006, Ischia, Italy, May 3-5, 2006*, pages 29–40.

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc.

Blum, N. (2015). Maximum matching in general graphs without explicit consideration of blossoms revisited. *CoRR*, abs/1509.04927.

Bogdanski, M., Lewis, P. R., Becker, T., and Yao, X. (2011). Improving scheduling techniques in heterogeneous systems with dynamic, on-line optimisations. In *Complex, Intelligent and Software Intensive Systems (CISIS), 2011 International Conference on*, pages 496–501.

Brodtkorb, A. R., Dyken, C., Hagen, T. R., Hjelmervik, J. M., and Storaasli, O. O. (2010). State-of-the-art in heterogeneous computing. *Sci. Program.*, 18(1):1–33.

Chen, J. and John, L. K. (2009). Efficient program scheduling for heterogeneous multi-core processors. In *Proceedings of the 46th Design Automation Conference, DAC 2009, San Francisco, CA, USA, July 26-31, 2009*, pages 927–930.

Choi, J., Singh, A., and Vuduc, R. W. (2010). Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2010, Bangalore, India, January 9-14, 2010*, pages 115–126.

Chung, E. S., Milder, P. A., Hoe, J. C., and Mai, K. (2010). Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs? In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 225–236.

Collange, S., Daumas, M., Defour, D., and Parello, D. (2010). Barra: A parallel functional simulator for GPGPU. In *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 351–360.

Craeynest, K. V., Jaleel, A., Eeckhout, L., Narváez, P., and Emer, J. S. (2012). Scheduling heterogeneous multi-cores through performance impact estimation (PIE). In *39th International Symposium on Computer Architecture (ISCA 2012), June 9-13, 2012, Portland, OR, USA*, pages 213–224.

Cui, Z., Liang, Y., Rupnow, K., and Chen, D. (2012). An accurate GPU performance model for effective control flow divergence optimization. In *26th IEEE International*

Parallel and Distributed Processing Symposium, IPDPS 2012, Shanghai, China, May 21-25, 2012, pages 83–94.

Dao, T. T., Kim, J., Seo, S., Egger, B., and Lee, J. (2015). A performance model for GPUs with caches. *IEEE Trans. Parallel Distrib. Syst.*, 26(7):1800–1813.

Dathathri, R., Reddy, C., Ramashekar, T., and Bondhugula, U. (2013). Generating efficient data movement code for heterogeneous architectures with distributed-memory. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, Edinburgh, United Kingdom, September 7-11, 2013*, pages 375–386.

Delimitrou, C. and Kozyrakis, C. (2013). Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pages 77–88.

Edmonds, J. (1965a). Maximum matching and a polyhedron with $0,1$ vertices. *J. of Res. the Nat. Bureau of Standards*, 69 B:125–130.

Edmonds, J. (1965b). Paths, trees, and flowers. *Canad. J. Math.*, 17:449–467.

Emani, M. K., Wang, Z., and O'Boyle, M. F. P. (2013). Smart, adaptive mapping of parallelism in the presence of external workload. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013, Shenzhen, China, February 23-27, 2013*, pages 13:1–13:10.

Eyerman, S. and Eeckhout, L. (2010). Probabilistic job symbiosis modeling for SMT processor scheduling. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010*, pages 91–102.

Fatahalian, K. (2008). From shader code to a teraflop: How shader cores work. http://s08.idav.ucdavis.edu/fatahalian-gpu-architecture.pdf.

Ghose, A., Dey, S., Mitra, P., and Chaudhuri, M. (2016). Divergence aware automated partitioning of OpenCL workloads. In *Proceedings of the 9th India Software Engineering Conference, Goa, India, February 18-20, 2016*, pages 131–135.

Grewe, D., Wang, Z., and O'Boyle, M. F. P. (2013). Portable mapping of data parallel programs to OpenCL for heterogeneous systems. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013, Shenzhen, China, February 23-27, 2013*, pages 22:1–22:10.

Guerreiro, J., Ilic, A., Roma, N., and Tomás, P. (2015). Multi-kernel auto-tuning on GPUs: Performance and energy-aware optimization. In *23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2015, Turku, Finland, March 4-6, 2015*, pages 438–445.

Han, H. (2014). Analyzing support vector machine overfitting on microarray data. In *Intelligent Computing in Bioinformatics - 10th International Conference, ICIC 2014, Taiyuan, China, August 3-6, 2014. Proceedings*, pages 148–156.

Hawkins, D. M. (2004). The problem of overfitting. *Journal of Chemical Information and Modeling*, 44(1):1–12.

Hong, S. and Kim, H. (2009). An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *36th International Symposium on Computer Architecture (ISCA 2009), June 20-24, 2009, Austin, TX, USA*, pages 152–163.

Hong, S. and Kim, H. (2010). An integrated GPU power and performance model. *SIGARCH Comput. Archit. News*, 38(3):280–289.

HPArch (2012). Macsim: A CPU-GPU heterogeneous simulation framework user guide http://comparch.gatech.edu/hparch/macsim/macsim.pdf.

Jablin, T. B., Jablin, J. A., Prabhu, P., Liu, F., and August, D. I. (2012). Dynamically managed data for CPU-GPU architectures. In *10th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2012, San Jose, CA, USA, March 31 - April 04, 2012*, pages 165–174.

Jablin, T. B., Prabhu, P., Jablin, J. A., Johnson, N. P., Beard, S. R., and August, D. I. (2011). Automatic CPU-GPU communication management and optimization. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 142–151.

Jia, H., Zhang, Y., Long, G., Xu, J., Yan, S., and Li, Y. (2012a). GPURoofline: A model for guiding performance optimizations on GPUs. In *Euro-Par 2012 Parallel Processing - 18th International Conference, Euro-Par 2012, Rhodes Island, Greece, August 27-31, 2012. Proceedings*, pages 920–932.

Jia, W., Shaw, K. A., and Martonosi, M. (2012b). Stargazer: Automated regression-based GPU design space exploration. In *2012 IEEE International Symposium on Performance Analysis of Systems & Software, New Brunswick, NJ, USA, April 1-3, 2012*, pages 2–13.

Jiao, Q., Lu, M., Huynh, H. P., and Mitra, T. (2015). Improving GPGPU energy-efficiency through concurrent kernel execution and DVFS. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2015, San Francisco, CA, USA, February 07 - 11, 2015*, pages 1–11.

Kaleem, R., Barik, R., Shpeisman, T., Lewis, B. T., Hu, C., and Pingali, K. (2014). Adaptive heterogeneous scheduling for integrated GPUs. In *International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24-27, 2014*, pages 151–162.

Karami, A., Mirsoleimani, S. A., and Khunjush, F. (2013). A statistical performance prediction model for OpenCL kernels on NVIDIA GPUs. In *The 17th CSI International Symposium on Computer Architecture Digital Systems (CADS 2013)*, pages 15–22.

Kavathekar, P. (2005). Maximum matching. https://www.cs.dartmouth.edu/reports/abstracts/2005.shtml.

Kerr, A., Diamos, G. F., and Yalamanchili, S. (2010). Modeling GPU-CPU workloads and systems. In *Proceedings of 3rd Workshop on General Purpose Processing on Graphics Processing Units, GPGPU 2010, Pittsburgh, Pennsylvania, USA, March 14, 2010*, pages 31–42.

Khronos (2016). OpenCL: The open standard for parallel programming of heterogeneous systems.

Kothapalli, K., Mukherjee, R., Rehman, M. S., Patidar, S., Narayanan, P. J., and Srinathan, K. (2009). A performance prediction model for the CUDA GPGPU platform. In *16th International Conference on High Performance Computing, HiPC 2009, December 16-19, 2009, Kochi, India, Proceedings*, pages 463–472.

Koufaty, D. A., Reddy, D., and Hahn, S. (2010). Bias scheduling in heterogeneous multi-core architectures. In *European Conference on Computer Systems, Proceedings of the 5th European conference on Computer systems, EuroSys 2010, Paris, France, April 13-16, 2010*, pages 125–138.

Lai, J. and Seznec, A. (2012). Break down GPU execution time with an analytical method. In *Proceedings of the 2012 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools, RAPIDO '12, 23 January, 2012, Paris, France*, pages 33–39.

Lakshminarayana, N. B., Lee, J., and Kim, H. (2009). Age based scheduling for asymmetric multiprocessors. In *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2009, November 14-20, 2009, Portland, Oregon, USA*.

Lang, J. and Rünger, G. (2014). An execution time and energy model for an energy-aware execution of a conjugate gradient method with CPU/GPU collaboration. *J. Parallel Distrib. Comput.*, 74(9):2884–2897.

Lee, M., Song, S., Moon, J., Kim, J., Seo, W., Cho, Y., and Ryu, S. (2014). Improving GPGPU resource utilization through alternative thread block scheduling. In *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014*, pages 260–271.

Leng, J., Hetherington, T., ElTantawy, A., Gilani, S., Kim, N. S., Aamodt, T. M., and Reddi, V. J. (2013). GPUWattch: Enabling energy optimizations in GPGPUs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 487–498, New York, NY, USA. ACM.

Li, S., Ahn, J. H., Strong, R. D., Brockman, J. B., Tullsen, D. M., and Jouppi, N. P. (2009). McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 469–480.

Li, T., Baumberger, D. P., Koufaty, D. A., and Hahn, S. (2007). Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing, SC 2007, November 10-16, 2007, Reno, Nevada, USA*, page 53.

Lucas, J., Lal, S., Andersch, M., Alvarez-Mesa, M., and Juurlink, B. (2013). How a single chip causes massive power bills GPUSimPow: A GPGPU power simulator. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 97–106.

Luk, C., Hong, S., and Kim, H. (2009). Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *42st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42 2009), December 12-16, 2009, New York, New York, USA*, pages 45–55.

Lutz, T., Fensch, C., and Cole, M. (2015). Helium: a transparent inter-kernel optimizer for OpenCL. In *Proceedings of the 8th Workshop on General Purpose Processing using GPUs, GPGPU@PPoPP 2015, San Francisco, CA, USA, February 7, 2015*, pages 70–80.

Margiolas, C. and O'Boyle, M. F. P. (2015). PALMOS: A transparent, multi-tasking acceleration layer for parallel heterogeneous systems. In *Proceedings of the 29th ACM on International Conference on Supercomputing, ICS'15, Newport Beach/Irvine, CA, USA, June 08 - 11, 2015*, pages 307–318.

Meenderinck, C. and Juurlink, B. H. H. (2008). (when) will CMPs hit the power wall? In *Euro-Par 2008 Workshops - Parallel Processing, VHPC 2008, UNICORE 2008, HPPC 2008, SGS 2008, PROPER 2008, ROIA 2008, and DPA 2008, Las Palmas de Gran Canaria, Spain, August 25-26, 2008, Revised Selected Papers*, pages 184–193.

Meng, J. and Skadron, K. (2009). Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs. In *Proceedings of the 23rd international conference on Supercomputing, 2009, Yorktown Heights, NY, USA, June 8-12, 2009*, pages 256–265.

Mierswa, I. (2007). Controlling overfitting with multi-objective support vector machines. In *Genetic and Evolutionary Computation Conference, GECCO 2007, Proceedings, London, England, UK, July 7-11, 2007*, pages 1830–1837.

Muck, T., Sarma, S., and Dutt, N. (2015). Run-DMC: Runtime dynamic heterogeneous multicore performance and power estimation for energy efficiency. In *2015 International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2015, Amsterdam, Netherlands, October 4-9, 2015*, pages 173–182.

Nagasaka, H., Maruyama, N., Nukada, A., Endo, T., and Matsuoka, S. (2010). Statistical power modeling of GPU kernels using performance counters. In *Green Computing Conference, 2010 International*, pages 115–122.

NVIDIA (2015). Cuda programming guide. http://docs.nvidia.com/cuda/pdf/cuda_c_programming_guide.pdf.

NVIDIA (2016). Nvidia OpenCL. https://developer.nvidia.com/opencl.

Pai, S., Thazhuthaveetil, M. J., and Govindarajan, R. (2013). Improving GPGPU concurrency with elastic kernels. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pages 407–418.

Pandit, P. and Govindarajan, R. (2014). Fluidic Kernels: Cooperative execution of OpenCL programs on multiple heterogeneous devices. In *12th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2014, Orlando, FL, USA, February 15-19, 2014*, page 273.

Pinedo, M. L. (2008). *Scheduling: Theory, Algorithms, and Systems*. Springer Publishing Company, Incorporated, 3rd edition.

Ravi, V. T., Becchi, M., Jiang, W., Agrawal, G., and Chakradhar, S. T. (2012). Scheduling concurrent applications on a cluster of CPU-GPU nodes. In *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2012, Ottawa, Canada, May 13-16, 2012*, pages 140–147.

Savage, N. (2016). Graph matching in theory and practice. *Commun. ACM*, 59(7):12–14.

Shelepov, D., Saez, J. C., Jeffery, S., Fedorova, A., Perez, N., Huang, Z. F., Blagodurov, S., and Kumar, V. (2009). HASS: a scheduler for heterogeneous multicore systems. *Operating Systems Review*, 43(2):66–75.

Sim, J., Dasgupta, A., Kim, H., and Vuduc, R. (2012). A performance analysis framework for identifying potential benefits in GPGPU applications. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 11–22, New York, NY, USA. ACM.

Singh, A. K., Shafique, M., Kumar, A., and Henkel, J. (2013). Mapping on multi/many-core systems: survey of current and emerging trends. In *The 50th Annual Design Automation Conference 2013, DAC '13, Austin, TX, USA, May 29 - June 07, 2013*, pages 1:1–1:10.

Snavely, A. and Tullsen, D. M. (2000). Symbiotic jobscheduling for a simultaneous multithreaded processor. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, pages 234–244, New York, NY, USA. ACM.

Song, S., Su, C., Rountree, B., and Cameron, K. W. (2013). A simplified and accurate model of power-performance efficiency on emergent GPU architectures. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 673–686.

Sun, E., Schaa, D., Bagley, R., Rubin, N., and Kaeli, D. R. (2012). Enabling task-level scheduling on heterogeneous platforms. In *The 5th Annual Workshop on General Purpose Processing with Graphics Processing Units, GPGPU-5, London, United Kingdom, March 3, 2012*, pages 84–93.

Tarakji, A., Gladis, A., Anwar, T., and Leupers, R. (2015). Enhanced GPU resource utilization through fairness-aware task scheduling. In *2015 IEEE TrustCom/BigDataSE/ISPA, Helsinki, Finland, August 20-22, 2015, Volume 3*, pages 45–52.

Ubal, R., Jang, B., Mistry, P., Schaa, D., and Kaeli, D. (2012). Multi2Sim: A simulation framework for CPU-GPU computing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 335–344, New York, NY, USA. ACM.

UIUC (2016). The LLVM compiler infrastructure project. http://llvm.org/.

van Werkhoven, B., Maassen, J., Seinstra, F. J., and Bal, H. E. (2014). Performance models for CPU-GPU data transfers. In *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2014, Chicago, IL, USA, May 26-29, 2014*, pages 11–20.

Venkat, A. and Tullsen, D. M. (2014). Harnessing ISA diversity: Design of a heterogeneous-ISA chip multiprocessor. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, pages 121–132.

Venu, B. (2011). Multi-core processors - an overview. *CoRR*, abs/1110.3535.

Wang, G., Lin, Y., and Yi, W. (2010). Kernel fusion: An effective method for better power efficiency on multithreaded GPU. In *2010 IEEE/ACM Int'l Conference on Green Computing and Communications, GreenCom 2010, & Int'l Conference on Cyber, Physical and Social Computing, CPSCom 2010, Hangzhou, China, December 18-20, 2010*, pages 344–350.

Wang, L., Huang, M., and El-Ghazawi, T. A. (2011). Exploiting concurrent kernel execution on graphic processing units. In *2011 International Conference on High Performance Computing & Simulation, HPCS 2012, Istanbul, Turkey, July 4-8, 2011*, pages 24–32.

Wang, Y. and Ranganathan, N. (2011). An instruction-level energy estimation and optimization methodology for GPU. In *Computer and Information Technology (CIT), 2011 IEEE 11th International Conference on*, pages 621–628.

Wende, F., Cordes, F., and Steinke, T. (2012). On improving the performance of multi-threaded CUDA applications with concurrent kernel execution by kernel reordering. In *Application Accelerators in High Performance Computing (SAAHPC), 2012 Symposium on*, pages 74–83.

Williams, S., Waterman, A., and Patterson, D. (2009). Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76.

Witten, I. H. and Frank, E. (1999). *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann.

Zakharenko, V., Aamodt, T., and Moshovos, A. (2013). Characterizing the performance benefits of fused CPU/GPU systems using fusionsim. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 685–688, San Jose, CA, USA. EDA Consortium.

Zhang, Y., Hu, X., and Chen, D. Z. (2002). Task scheduling and voltage selection for energy minimization. In *Proceedings of the 39th Design Automation Conference, DAC 2002, New Orleans, LA, USA, June 10-14, 2002*, pages 183–188.

Zhang, Y. and Owens, J. D. (2011). A quantitative performance analysis model for GPU architectures. In *17th International Conference on High-Performance Com-*

puter Architecture (HPCA-17 2011), February 12-16 2011, San Antonio, Texas, USA, pages 382–393.

Zhong, J. and He, B. (2014). Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling. *IEEE Trans. Parallel Distrib. Syst.*, 25(6):1522–1532.

Zhong, Z., Rychkov, V., and Lastovetsky, A. L. (2012). Data partitioning on heterogeneous multicore and multi-GPU systems using functional performance models of data-parallel applications. In *2012 IEEE International Conference on Cluster Computing, CLUSTER 2012, Beijing, China, September 24-28, 2012*, pages 191–199.